UNIVERSITY OF OSLO
Department of Informatics

Experimentally finding the right aggressiveness for retransmissions in thin TCP streams

How hard can it be?

Master thesis

Jonas Sæther Markussen

18. August 2014

Experimentally finding the right aggressiveness for retransmissions in thin TCP streams

Jonas Sæther Markussen

18. August 2014

Abstract

The traditional concept of fairness in TCP is based on being limited by congestion control. Today, however, we see that TCP is being used as transport for interactive applications that have latency requirements. These applications create application-limited, thin streams where retransmissions, rather than congestion control, is the factor controlling the performance of the flow. Keeping the maximum delay as low as possible is crucial to improving the Quality of Experience for interactive, thin-stream applications. As there is little existing work and no clear consensus on how time-dependent, thin-streams should be treated, we attempt to assess how aggressive these thin TCP streams can and should be on retransmission in order to reduce their retransmission latency when loss occurs. In this thesis, we discuss how we have created a Linux networking environment and conducted our experiments in order to find a reasonable trade-off between aggressiveness and fairness. Our findings indicate that an increased aggressiveness can be justified in competition with greedy streams, and we highlight some issues surrounding thin stream behaviour needs to be further studied.

Acknowledgements

What seemed like a relatively simple requests from my supervisors – set up a networking testbed and run some tests in order to assess how modifications to the congestion control for thin streams affect other traffic – turned out to be enough tears, late nights and outright desperation to be a topic for a thesis on its own. After much frustration and many dead-ends, accompanied by unexpected and, sometimes, weird results, as well as a recurring reminder that I did not understand network behaviour as well as I thought I did to begin with[1], was I finally able to turn my work into something that resembles a master's thesis.

First, I would like to thank my supervisors, Dr. Andreas Petlund and Dr. Carsten Griwodz, for their guidance and feedback on my work, patiently answering my questions, and taking time to review and discuss my observations and test results. Both of them have provided me with a lot information about new findings and results from the rest of the thin-stream group here at Simula as well as keeping me updated about research going on both at the networking group at the Department of Informatics at University of Oslo and by others working on the RITE project.

I would also like to thank Dr. Pål Halvorsen for his valuable input on my test results, and especially for helping me concretise my thesis and narrowing down my test scenarios to something that was manageable. He and Dr. Petlund really helped me tie together all the loose ends I had.

A special thanks to Bendik Rønning Opstad. Without his patching of `streamzero` and our combined efforts to fix and improve `analyseTCP`, my work simply wouldn't have been possible. I have also learned a great deal from our discussions on various topics and problems; everything from how the Linux kernel works, how TCP behaves and what is actually passed to the network driver and put "on the wire", to other issues such as best practices in software development and source control.

Additionally, I would like to thank Markus Fuchs and Dr. David Ros for their individual comparisons of their simulation results and my own observations from emulation, and especially for confirming my observations. I also want to thank them both for our discussions about differences and similarities in the observations as well as hypothesising about possible explanations, which gave me a lot of new ideas to try out.

My thanks also goes to Øystein Gyland for kick-starting my Linux networking skills, and getting me started on setting up my testbed, Preben Nenseth Olsen for a lot of tips on how to write a thesis, and to Tor Ivar Johannesen for getting me started with LaTeX as well as spending the entire summer in solitude with me out here on Fornebu, when everything was closed down for the summer holidays.

My gratitude is also due to Anders Moe and Christian Tryti for our friendly talks and frequent trips to the store. Thanks to Morten Ødegaard, Anders Ellefsen, Karoline Klever, Nora Raaum, Axel Sanner, Helene Cederstolpe Andresen, Chris Carlmar and Ståle Kristoffersen, as well as all my friends and family for believing in me and constantly nagging me to get my thesis done.

Finally, I would like to thank AK. This has been a stressful period in my life with a lot of work, and you were always there for me and supported me in so many ways without hesitation or asking for anything in return. You're always in my heart and I will never forget what you mean to me.

[1] ... and I probably still don't!

Contents

Experimentally finding the right aggressiveness for retransmissions in thin TCP streams **i**

Abstract **iv**

Acknowledgements **vi**

Contents **viii**

List of Figures **xiii**

List of Tables **xv**

List of Abbreviations **xvi**

1 Introduction **1**

1.1 Background and related work 1

1.2 Problem statement 2

1.3 Scope and limitations 2

1.4 Research method 3

1.5 Contributions 4

1.6 Outline 4

2 Thin streams **6**

2.1 Traffic characteristics 6

2.2 Latency requirements . . . 7
2.2.1 Online games . . . 8
2.2.2 Real-time multimedia applications . . . 8
2.2.3 Administrating remote systems . . . 9
2.2.4 High-frequency algorithmic trading . . . 9
2.3 Thin-stream transport protocols . . . 9
2.3.1 User Datagram Protocol . . . 10
2.3.2 Real-Time Transport Protocol . . . 10
2.4 Transmission Control Protocol . . . 11
2.4.1 Header layout . . . 12
2.4.2 Connection management . . . 14
2.4.3 Data transfer . . . 15
2.4.4 Flow control . . . 17
2.5 Congestion control in TCP . . . 19
2.5.1 Nagle's algorithm . . . 19
2.5.2 Delayed acknowledgements . . . 20
2.5.3 Congestion window . . . 20
2.5.4 Window algorithm variants . . . 24
2.5.5 Retransmission time-out calculation . . . 27
2.6 Thin-stream modifications to TCP . . . 28
2.6.1 TCP smart framing . . . 28
2.6.2 Early retransmit . . . 29
2.6.3 Modified fast retransmit . . . 30
2.6.4 Linear retransmission time-out . . . 30
2.6.5 Tail loss probe . . . 31
2.6.6 Redundant data bundling . . . 31
2.7 TCP fairness . . . 32

2.7.1 Max-min fairness . . . 33
2.7.2 Jain's fairness index . . . 34
2.8 Summary . . . 34

3 Experiment design and tools **36**

3.1 Metrics . . . 36
3.2 Design considerations . . . 37
3.2.1 Previous evaluations . . . 37
3.2.2 Simulation versus emulation . . . 39
3.2.3 Realistic packet loss . . . 40
3.2.4 Generating thin streams . . . 40
3.2.5 Choosing the network parameters . . . 41
3.3 Test environment . . . 42
3.3.1 Network topology . . . 43
3.3.2 Traffic control . . . 43
3.3.3 Router configuration . . . 45
3.3.4 Network emulator . . . 49
3.3.5 Traffic generation . . . 49
3.4 Analysis tools . . . 50
3.4.1 `tcpdump` . . . 50
3.4.2 `tcp-throughput` and `tput` . . . 50
3.4.3 `analyseTCP` . . . 51
3.4.4 `aqmprobe` . . . 55
3.4.5 `count3way` . . . 55
3.5 Summary . . . 56

4 Verifying the testbed **57**

4.1 A naïve approach . . . 57

4.1.1 Too congested? . . . 59

4.2 A thought-through approach . . . 60

4.3 Thin stream discrimination . . . 60

4.3.1 Comparing throughput and goodput . . . 60

4.3.2 Examining the loss rates . . . 63

4.3.3 Kernel buffering and repacketisation . . . 64

4.3.4 Thin stream clustering . . . 64

4.4 Summary . . . 65

5 Summarising the results **66**

5.1 Queue length evaluations . . . 66

5.2 Assessing the impact on other streams . . . 69

6 Conclusion **73**

6.1 Main contributions . . . 73

6.2 Future work . . . 74

Bibliography **75**

Internet Standards and Drafts **80**

Internet References **82**

A All test scenarios **87**

B Testbed configuration **88**

B.1 Specifications . . . 88

B.2 Topology . . . 89

C Web-site RTT measurements **90**

D Position paper **93**

D.1 On the Treatment of Application-Limited Streams 94

List of Figures

2.1 RTP encapsulation . . . 11

2.2 The TCP header . . . 12

2.3 Example of TCP header options . . . 14

2.4 Connection handling in TCP . . . 14

2.5 Retransmissions in TCP . . . 16

2.6 TCP sender-side buffer . . . 16

2.7 Cumulative acknowledgement in TCP . . . 17

2.8 Flow control in TCP . . . 18

2.9 TCP segment transmission time-line with and without Nagle's algorithm . . . 19

2.10 TCP slow start and congestion avoidance . . . 21

2.11 TCP fast recovery . . . 23

2.12 Fast recovery cycles in TCP Reno . . . 23

2.13 Congestion window growth stages . . . 26

2.14 Unused space in an Gigabit Ethernet frame . . . 31

2.15 Redundant data bundling . . . 32

2.16 Example network with six senders and two shared links . . . 32

3.1 Network configurations used in previous evaluations . . . 38

3.2 Testbed network topology . . . 43

3.3 Classes and qdiscs . . . 44

3.4 Packet enqueueing in a hierarchical qdisc configuration . . . 45

3.5 Token bucket algorithm for rate control . . . 46
3.6 Rate limitation accuracy at different clock granularities . . . 46
3.7 Packet flow through a Linux router with rate control . . . 47
3.8 Traffic capturing in our testbed . . . 50

4.1 Aggregated goodput of N thin streams (blue) competing with N greedy streams (red) . . . 58
4.2 Relative loss for N thin streams (blue) competing against N greedy streams (red) . . . 59
4.3 Goodput of N thin streams (blue) competing against 5 greedy streams (red) . . . 62
4.4 Comparing goodput and throughput for N unmodified thin streams competing against 5 greedy streams . . . 63
4.5 Relative packet loss for 30 thin streams and 20 greedy streams . . . 63

5.1 Relative byte loss using different qdiscs. 30 unmodified thin streams competing with 30 greedy streams. . . . 67
5.2 Impact of queue configuration on latency . . . 68
5.3 Throughput. 60 thin streams versus 10 greedy streams. . . . 70
5.4 Relative packet loss. 60 thin streams versus 10 greedy streams . . . 70
5.5 ACK latency. 60 thin streams versus 10 greedy streams . . . 71
5.6 ACK latency CDF annotation . . . 72

B.1 Testbed network topology . . . 89

List of Tables

2.1 Packet statistics for some interactive applications 7

2.2 TCP header control flags 13

3.1 RTT to the ten most popular web-sites 42

4.1 Number of problematic thin stream connections 59

4.2 Duration test, 2 hours and 5 minutes. Evaluated traffic: 40 thin streams; Cross-traffic: 10 greedy streams. 61

5.1 Test configuration used for our qdisc evaluation 67

B.1 Testbed system specifications 88

B.2 Testbed network specifications 88

C.1 Web-site RTT measurements 90

List of Abbreviations

`bfifo` byte-based FIFO.

`codel` controlled delay.

`cwnd` congestion window.

`htb` hierarchical token bucket.

`netem` network emulator.

`pcap` packet capture.

`pfifo` packet-based FIFO.

`prio` priority scheduler.

`red` random early detection.

`rwnd` receive window.

`sfq` stochastic fair queueing.

`ssthresh` slow start threshold value.

`tbf` token bucket filter.

`tc` traffic control.

ACK acknowledgement.

ADSL Asymmetric Digital Subscriber Line.

AIMD additive increase — multiplicative decrease.

ALD application-layer delay time.

ALG application-layer gateway.

API Application Programming Interface.

AQM active queue management.

ARP Address Resolution Protocol.

ARQ active repeat request.

BDP bandwidth-delay product.

BIC binary increase congestion control.

bps bits per second.

CDN content delivery network.

CPU Central Processing Unit.

DCE Direct Code Execution.

DNS Domain Name System.

DSACK duplicate selective acknowledgement.

DSCP DiffServ Code Point.

dupACK duplicate acknowledgement.

ECN Explicit Congestion Notification.

ER early retransmit.

F-RTO forward RTO recovery.

FACK forward acknowledgement.

FIFO first in — first out.

FIN finalize.

FTP File Transfer Protocol.

GB gigabyte.

Gbps gigabits per second.

GRO generic receive offload.

GSO generic segmentation offload.

HFT high-frequency algorithmic trading.

HOL blocking head-of-line blocking.

HTML5 HyperText Markup Language version 5.

HTTP HyperText Transfer Protocol.

IAT inter-arrival time.

ICE Interactive Connectivity Establishment.

ICMP Internet Control Message Protocol.

IETF Internet Engineering Task Force.

IP Internet Protocol.

IPv4 Internet Protocol version 4.

ISP Internet Service Provider.

ITT inter-transmission time.

ITU-T ITU Telecommunication Standardization Sector.

kB kilobyte.

kbps kilobits per second.

Kprobe kernel probe.

Kretprobe kernel return probe.

LAN Local Area Network.

LFN long fat network.

LRO large receive offload.

LT linear retransmission time-out.

Mbps megabits per second.

MFR modified fast retransmit.

MMOG Massively Multiplayer Online Game.

MSS Maximum Segment Size.

MTU Maximum Transfer Unit.

NAT Network Address Translation.

NIC network interface controller.

OS Operating System.

OSPF Open Shortest Path First.

OWD one-way delay time.

OWDVAR one-way delay variance.

P2P peer-to-peer.

PIF packets in flight.

qdisc queuing discipline.

QoS Quality of Service.

RDB redundant data bundling.

RDC Remote Desktop Connection.

RFC Request for Comments.

RTCP RTP Control Protocol.

RTO retransmission time-out.

RTP Real-Time Transport Protocol.

RTT round-trip delay time.

SACK selective acknowledgement.

SSH Secure Shell.

STUN Session Traversal Utilities for NAT.

SYN synchronize.

TCP Transmission Control Protocol.

TCP-SF TCP smart framing.

TFO TCP fast open.

TLP tail loss probe.

TSO TCP segmentation offload.

TURN Traversal Using Relays around NAT.

UDP User Datagram Protocol.

VNC Virtual Network Computing.

VoIP Voice over IP.

W3C World Wide Web Consortium.

WebRTC Web Real-Time Communications.

Chapter 1

Introduction

In the last couple of decades, we have seen a development of networking technology allowing a massive increase in capacity compared to the early days of the Internet. Following this increased capacity, the number of services provided by the Internet has exploded. Parallel to this trend of increased bandwidth usage, applications with real-time requirements have also emerged.

Today, we see that the Internet is used as a medium for a wide variety of interactive applications such as chat services, remote desktop, IP telephony, and networked games [1]. This has sparked renewed interest for research on these trends, and new attempts on reducing latency for interactive applications have emerged [2,3] [84,85]. Interactivity and time-dependency, however, has a multitude of requirements that are different from the requirements of applications that need high capacity.

1.1 Background and related work

There has been a lot of effort put into improving the Transmission Control Protocol (TCP) in order to achieve higher throughput while at the same time reacting to congestion building up in the network. These mechanisms are, however, designed on the assumption that TCP tries to consume as much as possible of the available bandwidth. This is not the case for traffic belonging to many interactive or time-dependent applications [4]. These kind of applications often produce network traffic that has a distinct pattern — low packet rate and small packet sizes. We call traffic flows with these characteristics *thin streams*.

As more of the traffic in the Internet is guided by application transmission patterns, rather than by congestion control, new patterns of behaviour are emerging. The effects of these developments are not well-understood. Studies have shown that the retransmission mechanisms of TCP produce higher retransmission delays for thin streams, as these mechanisms rely on many packets on the wire in order to be effective [4–8]. These studies suggests some modifications to the congestion control mechanisms in order to compensate for this unfortunate behaviour.

How fair these modifications are towards other traffic flows is something that has not been well investigated. Despite this, some of these modifications, particularly modified fast retransmit (MFR) and linear retransmission time-out (LT) [4], are already implemented the Linux kernel [9]. Although criticised, the concept of *fairness* between TCP streams is virtually synonymous with all streams having a fair share of the available bandwidth [10, 11] [49]. What we consider fair when streams with different requirements compete against each other is still an open question [12, 13].

1.2 Problem statement

The traditional fairness principle in TCP is that all streams sharing a limited resource, i.e. a network path with limited capacity, should receive an equal share of the bandwidth. In order to achieve this, TCP relies on its congestion control algorithm. This algorithm builds on the assumption that applications attempt to achieve the highest possible throughput. Interactive and time-dependent applications, however, do not attempt to send as much data as possible, but generate what we call thin streams — streams that are limited by the application rather than by congestion control. In a scenario where different TCP streams are competing over the same shared resource have different requirements, we need a different definition of fairness. The question remains: how aggressive can time-dependent traffic be in order to achieve the lowest possible latency?

The goal of this thesis is to determine if a more aggressive behaviour can be justified for an application-limited TCP stream in order to reduce overall latency. In particular, we attempt to experimentally answer how aggressive the retransmission mechanisms can be for thin streams while still remaining relatively fair towards other traffic.

1.3 Scope and limitations

Although there are a number of problems with using TCP for time-dependent data, such as head-of-line blocking (HOL blocking) and slow transmission rate ramp up, our focus is on investigating how aggressive thin streams can retransmit data as studies show that one of the largest factors contributing to increased latency is retransmission delays [4, 5]. We have limited ourselves to evaluating two thin stream modifications that are implemented in the Linux kernel — MFR and LT.

Network simulators are widely used in network research, but studies suggest that some simulators introduce uncertainties [13, 14]. Since the modifications we want to evaluate are already implemented in the Linux kernel, we decide to emulate a network using a Linux network testbed for our test environment. This test environment is created according to best practices [86] and our own findings, outlined in this thesis.

In order to assess the fairness of a more aggressive retransmission behaviour, we introduce competition over a shared, limited resource. In our case, this is a bottlenecked router. There are other forms of

competition that is worth investigating, but because of time constraints we limit ourselves to only consider the most common bottleneck: limited capacity. Even though Quality of Service (QoS) regimes are common in the Internet to improve the performance of time-dependent network traffic, we have not found time to explore them. Our test environment is therefore not optimally designed for thin-stream (or mixed-stream) traffic.

Due to time constraints, we have also limited ourselves to only focus on the most used TCP variant in Linux, CUBIC, using mostly default configurations. The default options for TCP in Linux are not optimised for time-dependent traffic, but we have experimented with some of the settings that we suspected could have an effect on thin streams.

Instead of using real interactive and time-dependent applications to generate network traffic, we use a program developed by the authors of the previous thin-stream studies to simulate network traffic with the same characteristics as traffic created by real applications.

1.4 Research method

We attempt to answer how aggressive thin-stream retransmissions can be by investigating how two existing thin-stream modifications, MFR and LT, affect greedy streams and other thin streams using unmodified retransmission mechanisms. To do so, we conduct a fairness experiment in a controlled test environment and analyse our findings.

As there is no unified definition of fairness when streams with different requirements compete, in order to confirm the validity of our findings, it is *imperative* that we subject network traffic to a realistic scenario and that we fully understand and trust our test environment to be correct. We design a Linux network testbed to perform our experiment in, and attempt to model reality as close as possible by choosing a network topology and setting network conditions that reflect a plausible, real-world scenario.

Three hypotheses were formulated based on the limitations surrounding TCP congestion control as shown by the previous thin-stream studies [4–6]:

> **Hypothesis 1.** *The thin-stream modifications to the TCP retransmission mechanisms improve the latency experienced by the application for application-limited streams when packet loss is caused by network congestion.*

This hypothesis serves the purpose of verifying the findings of the previous studies [4–6]. We want to evaluate how well the thin-stream modifications perform when congestion starts building up in the network.

> **Hypothesis 2.** *When competing over the same bottleneck, a more aggressive retransmission behaviour does not affect the performance of other TCP streams that are limited by congestion control (greedy streams).*

The authors of the previous studies argue that a more aggressive retransmission mechanism for application-limited streams can be justified since these streams do not contribute to congestion [4, 6, 15]. The TCP congestion avoidance algorithm is designed in such a way that the bandwidth share for each TCP stream should converge against an equal allocation. We test if the more aggressive retransmission behaviour leads to greedy TCP streams receiving a share that is less than fair.

> **Hypothesis 3.** *When competing over the same bottleneck, a more aggressive retransmission behaviour does not affect the performance of other TCP streams that are limited by the application (thin streams).*

Even if the bandwidth share of competing streams is only reasonably reduced and could still be considered fair, being more aggressive can still harm streams that are unable to recover from loss as quickly as greedy streams. In other words, we need to ensure that streams using the modifications do not affect other application-limited streams that use unmodified retransmission mechanisms. Since these streams are often generated by time-dependent applications, we expand our fairness consideration to include the experienced latency for this type of streams.

1.5 Contributions

To prove the hypotheses formulated in the previous section, as well as attempting to answer the question of how aggressive thin streams can be on retransmissions, we performed work consisting of analysis, implementation and evaluation. Our main contributions are listed here:

- Evaluation of the MFR and LT retransmission modifications, showing that application-limited streams using the modifications are still at a disadvantage when sharing a resource with greedy streams. This suggests that an even more aggressive behaviour can be justifiable.
- Implementation of a deterministic network testbed and discussions of various pitfalls and challenges surrounding experimentally finding the appropriate aggressiveness for thin streams.
- Improvements to the `pcap` analysis tool, `analyseTCP`, originally created by the authors of the previous thin-stream studies [4, 15], as well as extending the functionality and implementing new features.
- Implementation of a kernel module that attaches itself to a Linux router queue and provides accurate drop and queue statistics.

1.6 Outline

This thesis attempts to assess the right aggressiveness for retransmissions in TCP thin streams, from the design of a fairness experiment to the analysis of the results and an evaluation of the observed behaviour.

- **Chapter 2** describes the characteristics of interactive data traffic and defines the concept of thin streams. We also outline modus operandi for TCP and how the mechanisms for reacting to network congestion works. We explain how these mechanisms cause extra application latency for thin-stream applications. Finally, we introduce the thin-stream modifications we evaluate and give a brief explanation of how they work.

- **Chapter 3** discusses the design and implementation of an experiment to test the TCP friendliness of the thin-stream modifications. Here we introduce a network testbed for performing our tests in order to evaluate how the thin-stream modifications affect other network traffic streams. We also outline which parameters we have decided to focus on, and the criteria we have chosen for considering the fairness.

- **Chapter 4** describes our iterative process to find sensible test parameters in order to conduct our fairness experiment. We discuss some unexpected observations, and describe how we investigated them as part of our effort to verify the validity of our findings. We also review some of the properties of application-limited streams and how their behaviour is unfortunate in a congested network, and outline the various factors that come into play in an attempt to explain our observations.

- **Chapter 5** presents the results from our tests as well as a discussion of some of the observed effects.

- **Chapter 6** concludes this thesis by summarising our findings and experiment results and what we learned from these. Finally, we outline some candidate topics for further research and investigations.

Chapter 2

Thin streams

The requirements for data delivery have changed significantly since the Internet's infancy. In the early days of the Internet, programs were designed around simple *client-server* models, focused around applications like web access, file transfer and e-mail. In the last decade, there has been an increase in the demand for more bandwidth, as well as a shift in traffic patterns to more complex *peer-to-peer (P2P)* models, with focus on both uploading as well as downloading, suitable for file sharing services such as BitTorrent. More recently, we have also seen a large growth in the number of streaming services, such as Netflix, YouTube and Spotify. [1]

Today, we see a use of the Internet that reflects aspects of real life. We interact in virtual environments, control remote systems and hold audio/video conferences. These interactive applications introduce a multitude of requirements to the infrastructure, since the experienced latency profoundly impacts the perceived quality of the service. Real-time computing systems, such as trading robots or sensor networks, are also a type of applications that require low latency [16–18].

On modern computer systems, the arguably largest source of latency is network traffic delays. This is due to the *best-effort* design of the Internet Protocol (IP) [50]; a design that paradoxically made it so popular for interconnecting computer networks in the first place. IP networks are packet-switched and uses routing algorithms in order to be fault tolerant. However, this design choice also means that datagrams, or *packets* as they are commonly called, are sometimes lost, delayed, corrupted or could arrive in a different order than they were sent. Later in this chapter, we will discuss protocols and mechanisms that aims to make the Internet more reliable.

2.1 Traffic characteristics

Network traffic generated by time-dependent applications, especially by those that are interactive, have distinct characteristics. Table 2.1 shows a selection of packet statistics collected from a range of time-dependent applications [4, 15]. It shows the distribution of packet payload sizes, packet inter-arrival time

	Application	Payload size (B)			Inter-arrival time (ms)						Bandwidth (avg)	
		min	avg	max	min	avg	med	max	1%	99%	pps	Kbps
Thin	Anarchy Online	8	98	1333	7	632	449	17032	83	4195	1.6	2.2
	Age of Conan	5	80	1460	<1	86	57	1375	24	386	11.6	12
	BZFlag	4	30	1448	<1	24	<1	540	<1	151	41.7	31
	Halo 3 (8 players) (UDP)	32	247	1264	<1	36	33	1403	32	182	27.8	60
	Halo 3 (6 players) (UDP)	32	270	280	32	67	66	716	64	69	14.9	36
	Test Drive Unlimited (UDP)	34	80	104	<1	40	33	298	<1	158	25	23
	Tony Hawk's Project 8 (UDP)	32	90	576	<1	308	163	4070	53	2332	3.2	5.8
	World in Conflict (server)	4	365	1361	<1	104	100	315	<1	300	9.6	31
	World in Conflict (client)	4	4	113	16	105	100	1022	44	299	9.5	4.4
	World of Warcraft	6	26	1228	<1	314	133	14855	<1	3785	3.2	2
	Skype (2 users) (UDP)	11	111	316	<1	30	24	20015	18	44	33.3	37
	Skype (2 users) (TCP)	14	236	1267	<1	34	40	1671	4	80	29.4	69
	RDC (Windows)	8	111	1417	1	318	159	12254	2	3892	3.1	4.5
	SSH (text session)	16	48	752	<1	323	159	76610	32	3616	3.1	2.8
	VNC (client)	1	8	106	<1	34	8	5451	<1	517	29.4	17
	VNC (server)	2	827	1448	<1	38	<1	3557	<1	571	26.3	187
Greedy	YouTube	112	1446	1448	<1	9	<1	1335	<1	127	>1000	1.3K
	HTTP download	64	1447	1448	<1	<1	<1	186	<1	8	>1000	14K
	FTP download	40	1447	1448	<1	<1	<1	339	<1	<1	>1000	82K

Table 2.1: Packet statistics for some interactive applications

(IAT) and average bandwidth consumption in packets per second and kbps. Statistics for three non-interactive applications have also been included for comparison, and these are shown in the last three rows.

As shown in the results, the last three applications generate network traffic that attempts to consume as much of the available bandwidth as they can get and are limited by congestion control. Traffic flows that behave this way are commonly known as *greedy* streams. On the other hand, we see that the traffic generated by the time-dependent applications have a high packet IAT and a small average packet size. In other words, these non-greedy streams are application limited. We call flows with these characteristics *thin* streams.

A third characteristic that also is quite evident, is of course variance in IAT. We will explain the impact of having regular packet transmission intervals as opposed to varying IAT in chapter 4.

2.2 Latency requirements

To understand why thin streams have the traffic pattern they have, we need to look to the latency requirements of these applications. A large majority of thin-stream applications involve user interaction, either between users or with a remote system. From this we argue that, generally speaking, the intensity of the interaction determines the requirements to the application latency and how the experienced quality of service is affected under non-optimal conditions. In this section we describe various real-time applications and their requirements.

2.2.1 Online games

In the last decade, we have seen so-called Massively Multiplayer Online Games (MMOGs) exploding in popularity [1, 19]. Fast paced and high precision games require lower latencies than games with relatively slower interaction. Studies suggest that the threshold for tolerated latencies is around 100 ms for first-person shooter games, 500 ms for action role-playing games and 1000 ms for real-time strategy games [20–22]. However, professional gamers can have an actions per minute count of over 300 on average during a game [87]. This suggests that the tolerated thresholds for some real-time strategy games would be closer to that of first-person shooter games.

We can see from table 2.1 that the high intensity games, such as the first-person shooter games, have slightly lower IAT than the role playing games. This is most likely due to the fact that first-person shooters demand a higher frequency of position updates in order to be perceived as reliable. With too much delay for position updates, the user experiences in-game lag, and objects in the game can move erratically which contributes to a bad gaming experience [23].

2.2.2 Real-time multimedia applications

Audio/video conferencing software, IP telephony, Voice over IP (VoIP) systems, etc., are other examples of thin-stream applications. Today, we see that software like Microsoft Lync, Skype, TeamSpeak, TeamViewer and other virtual collaboration tools are becoming increasingly popular in workplaces as well as in homes.

Many IP telephony systems use the G.7xx series of audio compression algorithms recommended by ITU-T [88]. For example, G.711 and G.729 have bandwidth requirements of 64 and 8 kbps respectively, and the packet payload size is determined by the packet transmission cycle. For G.711, this cycle is typically a few tens of milliseconds, resulting in packet sizes that are between 80 and 320 bytes [15,24]. Similarly, the traffic generated by Skype as seen in table 2.1, has packet payload sizes that average on 236 bytes (TCP) and have a bandwidth usage of around 69 kbps.

Web Real-Time Communications (WebRTC) is a relatively new framework, developed jointly by IETF and W3C and being supported by Google, Opera and Mozilla. [89]. It aims to be a simple HTML5 Application Programming Interface (API) for web developers, enabling real-time P2P communications in the browser. It uses multiple codecs and compression algorithms, such as Opus [51] [90] (which incorporates SILK originally developed for Skype), G.711, G.722, iLBC [52], iSAC, as well as VP8 for video. WebRTC uses the Real-Time Transport Protocol (RTP) for transport protocol. This means that it uses Interactive Connectivity Establishment (ICE), Traversal Using Relays around NAT (TURN), and Session Traversal Utilities for NAT (STUN) [53–55], in addition to RTP-over-TCP, for Network Address Translation (NAT) and firewall traversal.

For voice communications, ITU-T defines the acceptable (average) one-way transmission delay to be around 150-200 ms, with 400 ms as the absolute maximum [25] [91] and that a gap between audio and video (lip sync) of 45-100 ms is acceptable [26] [92, 93]

2.2.3 Administrating remote systems

Another popular type of interactive application is controlling and administrating systems remotely. Applications of this type usually send instructions and commands to the remote system, either periodically or in an event-based manner. As many of these systems operate in real time, they need to be able to react quickly to control signals.

Four popular applications for this are Telnet, Secure Shell (SSH), Remote Desktop Connection (RDC) and Virtual Network Computing (VNC). As seen in table 2.1, these applications create packets that are small and have a high average IAT. This is caused by the fact that network traffic is generated when the user interacts with the system, e.g. presses keys on his keyboard or moves the mouse cursor, but the user interaction is not as intensive as with games described in section 2.2.1.

Studies show that the average computer user can type 33 words per minute on average when copying a transcript, while a skilled user can type 87 words on average [15] [94]. Using a standardized word count of five characters per word [27], this means that the average user can type 2.75 characters per second on average and the skilled user up to an average of 7.25 characters per second. This means that 363 and 137 milliseconds will pass between each typed letter for the average and skilled user respectively.

2.2.4 High-frequency algorithmic trading

High-frequency algorithmic trading (HFT) is a field within real-time computing that is becoming increasingly popular. In 2010, it was estimated that HFT accounted for 56% of the entire equity turnover in the U.S. [28]. This has led to a technological arms race between HFT firms, with each firm trying to be faster and smarter than the others.

While many HFT firms use proprietary algorithms in order to outperform their competitors, they also have a need for speed. A study from 2010 shows that the speed of light[1] has become the bottleneck preventing HFT traders to operate on a global level [29], so firms often implement their systems as close to stock exchanges as possible [16]. Many of these systems have to react to events at sub-millisecond speed [16–18]. Estimates suggests that even a 1 millisecond improvement in latency for every transaction could be worth up to $100 million a year to a large investment bank or a broker firm [16–18] [95, 96].

2.3 Thin-stream transport protocols

Although the scope of this thesis is related to problems surrounding TCP as transport protocol for thin streams, there exist transport protocols that are designed for network transport of interactive and latency sensitive data. Many of these protocols are, however, not as well understood as TCP and User Datagram Protocol (UDP), and the general support for them in various networking components is uncertain. [4, 30, 31]

[1] Light propagates through 1 kilometre of fiber in approximately 3 microseconds.

2.3.1 User Datagram Protocol

UDP [56] is one of the simplest transport protocols, but it is nonetheless widely used for multimedia and interactive applications where some data loss is considered acceptable. It is a stateless, connectionless and message-oriented transport protocol, where a program can send messages (datagrams) to other hosts over an IP network without any transmission channels or data paths set up. It maps directly to the unreliable nature of IP; UDP offers no guarantee of delivery or ordering. Its simplicity makes it attractive for real-time applications where error correction is not necessary or can be done on the application layer. and where losing data (dropping a packet) is preferable to waiting for it to be retransmitted. [30, 31]

However, because of the IPv4 address space exhaustion[2], home consumer devices are almost exclusively connected to private networks behind NAT gateways. A common NAT implementation is the mapping of program ports to actual hosts on the private network, but this proves difficult with a unidirectional connection-less protocol like UDP [30]. Because of this, most home consumer oriented UDP-based applications offer TCP as a fall-back protocol, since TCP is bidirectional and the host behind NAT can initiate the connection instead. [4].

2.3.2 Real-Time Transport Protocol

RTP [57, 58] is a generic protocol suitable for transporting real-time data, and is used extensively in media streaming and VoIP implementations, together with its sibling protocol RTP Control Protocol (RTCP). [30]

It is not really a transport-layer protocol, but is positioned in between the application layer and the transport layer in the protocol stack. This means that the RTP interface is through a user-space RTP implementation, and this interface is responsible for multiplexing the streams and use a suitable transport-layer protocol to transmit the RTP packets. RTP has no built-in mechanism for retransmissions, reliability or congestion control. It offers facilities useful for multimedia, such as timestamping and sequence numbering, which allows for buffering and synchronizing multiple streams, e.g. an audio and video stream.

RTP was designed to transport a wide variety of multimedia formats, and allows a format or encoding to be specified through *RTP profiles*. A profile defines the codec used to encode the payload data, and an RTP packet can contain multiple encodings of the same data [30].

Figure 2.1 illustrates how RTP packets are encapsulated. Most RTP implementations use UDP, but it is possible to send RTP over other transport protocols, such as TCP. This means that RTP is at the mercy of the behaviour and treatment of the transport-layer protocol it uses.

[2] `http://www.ripe.net/internet-coordination/ipv4-exhaustion`

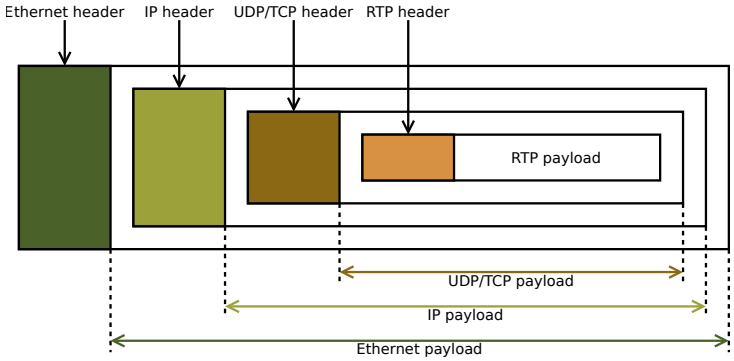


Figure 2.1: RTP encapsulation

2.4 Transmission Control Protocol

TCP, defined by RFC 793 [59] and updated by RFC 1122 [60], is a data transport protocol specifically designed to provide a reliable, end-to-end data delivery service over an unreliable data forwarding plane. It complements the best-effort, connection-less packet-switching design of IP networks by adding means for reliable, connection-oriented in-order data transmission, flow control and adaptiveness to network congestion. It is considered one of the core protocols of the Internet protocol suite, and the Internet today is in fact so dependent on TCP, that the suite is often referred to as TCP/IP [30, 31] [97].

TCP will establish, maintain and tear down a full duplex, reliable byte stream between two processes running on separate hosts over an interconnected network. Seen from the application, this byte stream works like a pipe, where data (bytes) go in at one end and arrives at the other, in-order and error free, thus making the underlying architecture transparent[3]. [59, 61] [32]

TCP divides chunks of bytes into **segments**. These segments consist of a 20-byte fixed header (plus an optional part) followed by zero or more data bytes. Segments are encapsulated in, and sent as, IP datagrams, thus the segment's length is restricted by the maximum IP payload size (65,515 bytes [50]) in addition to the network Maximum Transfer Unit (MTU). which generally is 1500 bytes — the same as the Ethernet payload size [30]. This means that the *Maximum Segment Size (MSS)*, the size of the largest possible packet that will not be fragmented, is $MSS = MTU - IP\ header\ length - TCP\ header\ length$. TCP segments are essentially the same as *packets* or *datagrams*, and the names are used interchangeably throughout the literature, even though strictly speaking "packets" would refer to packets from an IP layer point-of-view, while "datagrams" would be packets with data payload (i.e. UDP datagrams). We will use *segments* when talking about TCP packets, with or without data payload.

Since segments can be lost, e.g. due to network congestion, TCP uses retransmission to ensure reliability. Each byte is given a sequence number. When a sender transmits a segment, it gives the segment a sequence number corresponding to the first byte in that segment and starts a timer. The receiver explicitly

[3] In this context, transparency means that the running program only sees a *handle* (socket, see 2.4.2) provided by TCP to this byte stream "pipe". This handle has a simple API: *connect*, *write*, *read* and *close*. The "gory details" of the network is hidden from the program. Please refer to the literature for more details. [30] [59] [98]

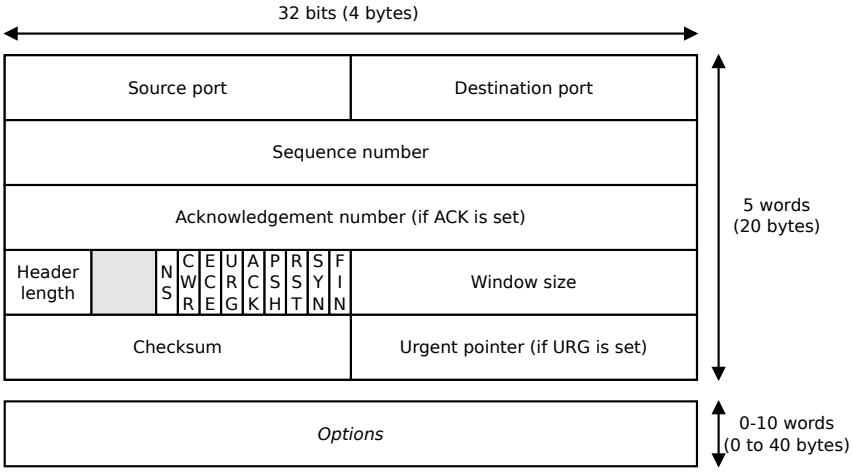


Figure 2.2: The TCP header

notifies the sender about a received segment by sending back an **acknowledgement (ACK)**, accounting for the highest byte in the received segment. If a segment remains unacknowledged after the timer goes off, the segment is assumed lost and retransmitted (until acknowledged by the receiver). Segments can also arrive out of order, e.g. due to routing changes, so to ensure in-order delivery to the application, TCP uses a sliding window algorithm with cumulative acknowledgement, which is described in section 2.4.3.

2.4.1 Header layout

Figure 2.2 shows the TCP segment header, which every TCP segment begins with. The header may be followed by a set of header options. After the options, if any, the data bytes may follow, but segments without any payload is allowed and is commonly used for ACKs and other control messages. The segment header structure is, as defined in RFCs 793, 1122, 3168 and 3540 [59, 60, 62, 63], as follows:

Source port (16 bits): Identifies the sending port.

Destination port (16 bits): Identifies the receiving port.

Sequence number (32 bits): The sequence number of the first data byte of this segment (every byte is given a sequence number in a TCP stream). The first sequence number is chosen randomly by the host initiating the connection, so it does not necessarily start on 1.

Acknowledgement number (32 bits): The sequence number of the next byte expected (not the last correctly received).

Header length (4 bits): Specifies how long the header is, in 32-bits words (5 = no options). Maximum value is $2^4 - 1 = 15$ words, 40 bytes of options.

Reserved (3 bits): For future use, and should be set to zero.

Flags (9 bits): Nine control bits, see table 2.2.

Window size (16 bits): The size of the receive window; the number of bytes the sender of this segment is willing to receive (see 2.4.4).

Checksum (16 bits): Checksum for data integrity. Should add up to 0.

Urgent pointer (16 bits): See the description for the URG flag below.

Table 2.2 shows the nine different control bits, or "flags", used by TCP. NS, CWR and ECE are used for a congestion notification technique called Explicit Congestion Notification (ECN) [62]. If the URG flag is set, this field indicates a byte offset from the current sequence number at which urgent data in the segment begins. It can be used to transfer out-of-band interrupt style signalling data, for applications like Telnet [64]. The ACK flag indicates that the packet contains an acknowledgement, all packets after the initial SYN packet should have this set. The PSH flag is used for quick transmission, indicating that data in buffers should be delivered to the application. In other words, data in sender-side buffers should be sent at once, while data in receiver-side buffers should be "pushed" to the application at once. The RST, SYN and FIN flags are used to reset, synchronize (initiate) and terminate a connection respectively. [30, 31] [99]

As specified in RFC 793 [59], we can have a variable amount of additional options after the initial, fixed-size segment header. Each TCP option is a three field value: Option kind (1 byte), option length (1 byte) and option data (variable length). Figure 2.3 shows how an example of TCP option layout in the options field with three options. Note that option 1 and 3 are padded to align with 32-bits.

Three commonly used TCP options are selective acknowledgement (SACK) [65], receiver window scaling and TCP timestamp [66]. A TCP sender sending its timestamp in headers are used for improved round-trip delay time (RTT) measurement and for better assessing when the sequence number has wrapped. SACK is further explained in 2.4.3, and window scaling is explained in 2.4.4.

The TCP header and IP header may be compressed, as proposed in RFC 1144 [67].

NS	ECN nonce concealment protector
CWR	Congestion window reduced
ECE	ECN echo
URG	Urgent pointer is significant
ACK	Acknowledgement number is significant
PSH	Push buffered data to the receiving application
RST	Reset the connection
SYN	Synchronize sequence numbers
FIN	No more data from sender (finalize)

Table 2.2: TCP header control flags

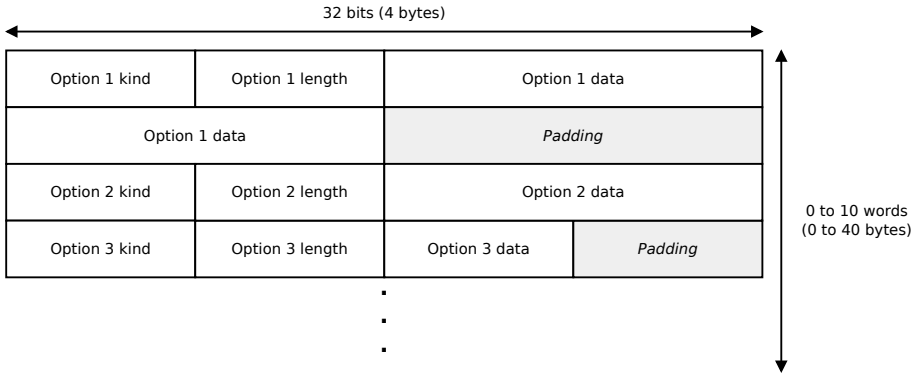
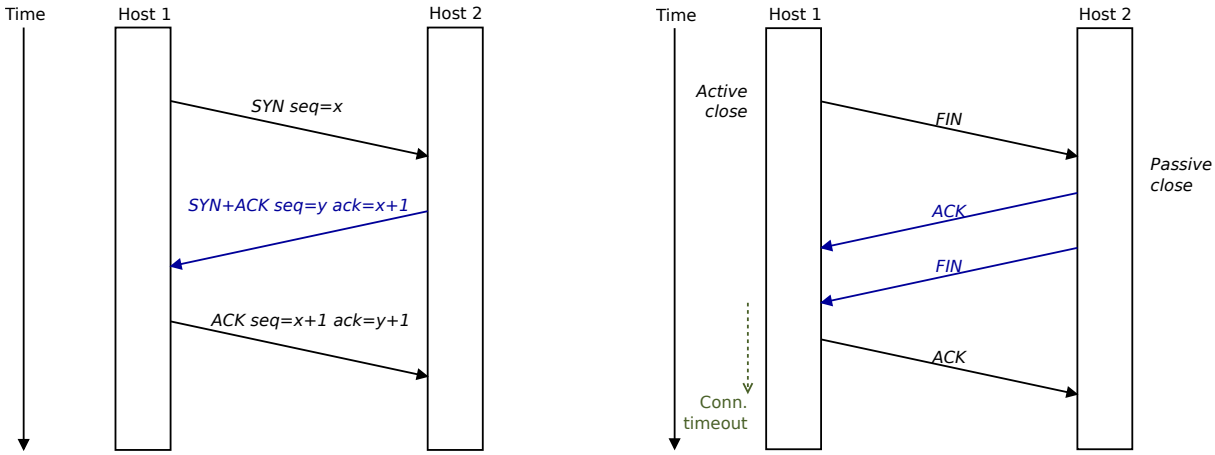


Figure 2.3: Example of TCP header options



(a) Establish (open connection)

(b) Finalize (close connection)

Figure 2.4: Connection handling in TCP

2.4.2 Connection management

TCP establishes a connection-oriented byte stream between two processes, either on the same machine or two different hosts. The hosts are uniquely identified by the network layer addressing scheme, i.e. IP addresses. A process is in turn identified by a port number. An address and a port number forms what is called a socket, and a pair of sockets uniquely identifies a TCP connection. This allows multiple processes within a single host to use TCP simultaneously, and a socket may be used for multiple streams [59]. In other words, services, such as HyperText Transfer Protocol (HTTP) and File Transfer Protocol (FTP), can be bound to well-known ports, and many clients can connect to the same service on the same server [68] [30].

Figure 2.4 shows how TCP establishes and tears down a connection. The connection is established using a three-way handshake. The initiator, Host 1, sends a segment with the SYN flag set, Host 2 replies with a segment with both the SYN and ACK flag set, and finally Host 1 acknowledges the active connection with an ACK. Note that the sequence number does not necessarily begin at 1. A TCP sender will choose a random starting number [30, 31]. This is a security measure to prevent a malicious user from guessing

the ACK number and leaving the receiver in a half-open state [30].

In addition to synchronizing the sequence numbers, by setting the sequence number and acknowledgement number as shown in fig. 2.4a, other tasks may be performed, such as determining the MSS (RFC 1191 [69]) or scaling the window size (RFC 1323 [66]). A proposed extension called TCP fast open (TFO) improves performance by allowing data to be sent with the opening SYN segments [100, 101].

Tearing down a connection can be done in different ways depending on the implementation, either by a four-way handshake — where both sides terminate the connection independently as shown in fig. 2.4b — or by a three-way handshake. The three-way handshake is similar to the four-way, but where both the ACK and FIN flags are set for the same segment. The latter is perhaps the most common method [99].

There are some issues with the three-way-handshake and connection tear-down: Delayed segments that arrive after the connection is terminated may lead to confusion if there are any subsequent streams, i.e. the receiver might think that the segments belongs to the current rather than the previous connection. So-called "half open" connections — where one side has terminated while the other has not — may also occur if some of these segments are lost. A more detailed explanation of connection handling in TCP can be found in literature [30] and in RFC 793 [59]. Stream termination and implementation considerations are further explained in RFC 1122 [60].

2.4.3 Data transfer

As mentioned previously in this chapter, every byte belonging to a TCP stream is given its own sequence number, and chunks of bytes are sent as segments. Segments are explicitly acknowledged by the receiver, by setting the ACK flag and the acknowledgement number to the *next expected byte*. Each segment header has both a sequence number and an acknowledgement number, allowing data to be transferred both ways (full duplex). In other words, ACKs can piggy-back on data segments, or they can be sent as stand-alone segments if there is no data to send back (packets containing only the header, zero-length payload).

If a segment is lost somewhere along the line from sender to receiver, it has to be retransmitted. Essentially, when TCP sends a segment it also starts a timer for that segment. If an ACK is received acknowledging the segment, the timer is cancelled and the next segment is sent. If the timer times out and no ACK for that segment is received, the segment is re-transmitted. How this time-out interval is calculated is explained in section 2.5.5.

The capability of retransmitting lost segments is achieved through buffering. The sender temporarily stores unacknowledged segments in a buffer. Figure 2.5 shows how this works. For simplicity, the sequence and acknowledgement numbers is the same as the segment number, while in reality, it would be a byte count. Host 1 transmits first segments 0-3, but segment 2 is lost (or massively delayed) somewhere along the line. The receiver sends an ACK every time it receives a segment, but since it has not received segment 2, it can not accept segment 3, and therefore replies that segment 2 is still the next expected

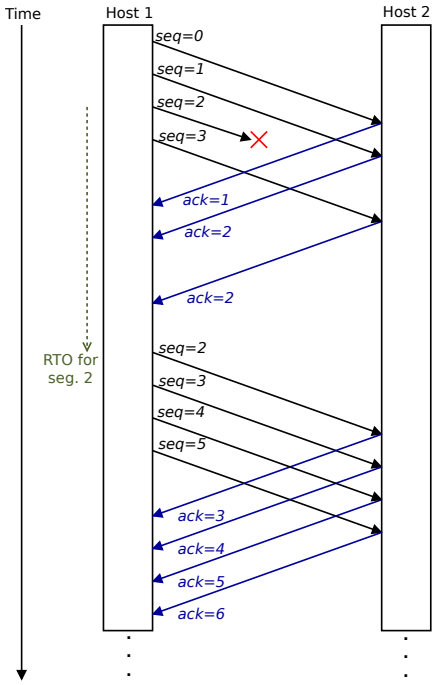
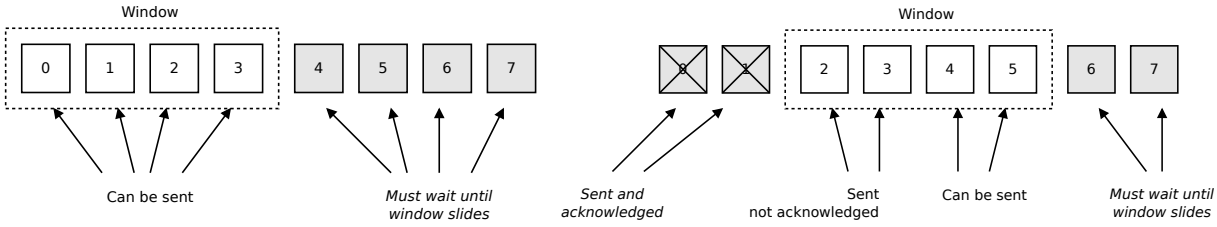


Figure 2.5: Retransmission in TCP



(a) Initial window

(b) After the time-out

Figure 2.6: TCP sender-side buffer

segment. Segment 2 (and 3) is retransmitted after its time-out. Note that the sender does not send one segment and wait for an ACK and then sends the next, but rather sends an entire *window* of segments. This is what is called a sliding window protocol, and is explained further in section 2.4.4.

However, transmitted segments can be lost both ways — in other words, acknowledgements may also be lost. To prevent unnecessary retransmissions if an ACK is lost, TCP uses a mechanism called **cumulative acknowledgements**. Figure 2.7 illustrates this. The ACK for segment 2 (ack=3) is lost, but the ACK for segment 3 (ack=4) is received before the time-out for segment 2 goes off. The sender then knows that the segment was actually received, thus no retransmission of that segment is required.

The combination of the sliding window and cumulative ACKs is essentially what is called a "Go-Back-N" active repeat request (ARQ). Although simple to implement, "Go-Back-N" is not very efficient. Even though the receiver may choose to buffer out-of-order segments that are within the window size — which in the example in figure 2.5 would be segment 3 — with cumulative ACKs, the sender can not

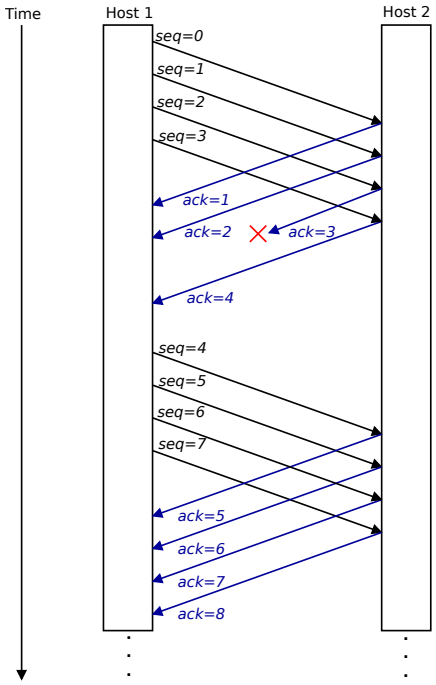


Figure 2.7: Cumulative acknowledgement in TCP

know whether it has been received or not, and thus has to resend it.

This is addressed by RFC 2018 [65], which makes it optional for TCP hosts to implement **SACK**[4]. Similar to the "Selective Repeat" ARQ [30], the receiver explicitly tells the sender which segments were lost and which were received. This is done by using header options for reporting information about received segments back to the sender. [70]

RFC 2883 [71] extends SACKs even further, by introducing duplicate selective acknowledgement (DSACK), which allows the receiver to notify the sender about the sequence numbers of the segments that triggered the ACK, thus allowing the sender to infer the order of segments received and from this understand when it has unnecessarily retransmitted a packet. The sender uses this to (better) assess the network loss.

2.4.4 Flow control

In addition to offering reliability (in form of explicit acknowledging received data), TCP also has mechanisms to manage the data rate between a TCP sender and a receiver. Although they somewhat overlap, it is important to distinguish *flow control* from *congestion control*. The latter is explained section 2.5.

As mentioned in section 2.4.3, a TCP sender transmits an entire window of segments rather than transmitting one segment at the time. The size of this window is set by the receiver, by setting the "window size field" in the ACK segment header, and the value in this field is interpreted by the sender as how many bytes the receiver is willing to accept next. This is what is called the **receive window (`rwnd`)**. [72]

[4] A study from 2011 on Google web servers shows that 96% of all connections supported SACK [7]

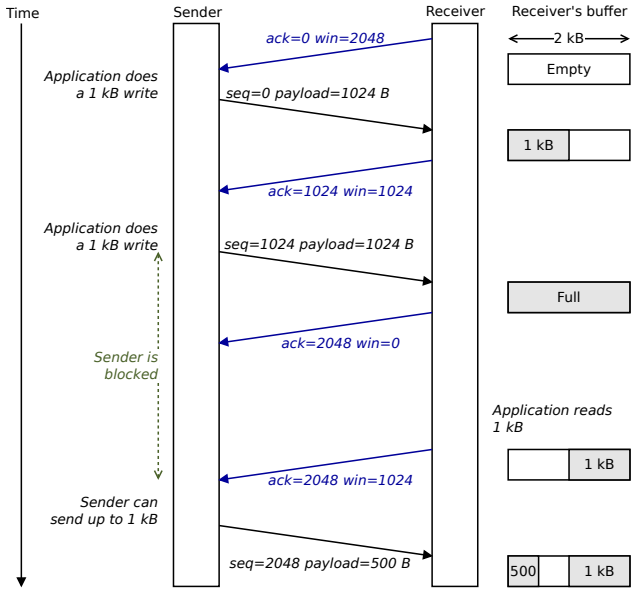
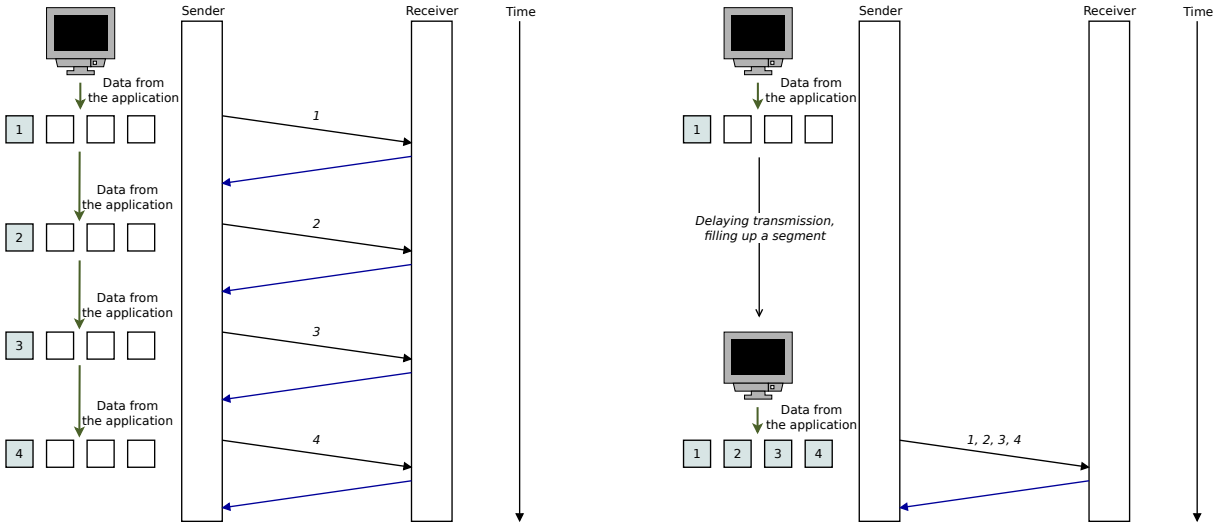


Figure 2.8: Flow control in TCP

In figure 2.5, `rwnd` is $4 \times \texttt{MSS}$. When the time-out for segment 2 occurs, the sender can transmit segment 3-5 as well, since the successful arrival of segment 0 and 1 means that the window "slides" two segments. Figure 2.6 shows how this works. A more detailed explanation of the sliding window and "Go-Back-N" ARQ mechanism can be found in literature. [30, 31]

By adjusting `rwnd` and setting the window size corresponding to the available receive buffer, a TCP receiver can notify the sender that it is currently being overwhelmed and that the sender needs to slow down the transmission rate. Figure 2.8 outlines how this work. While the window size is 0, the sender may not send any segments, but there are two exceptions: urgent data may be sent — for example so that the user can kill a remote application — and an empty segment can be sent so it forces the receiver to re-announce the acknowledgement number and window size. This is to prevent deadlocks should window size announcements be lost. [30]

When a network has a large bandwidth-delay product (BDP), the maximum possible window size becomes too small: $2^{16} = 65,535$ bytes (≈ 64 kB) is simply not large enough to accommodate the bandwidth of these high-capacity links. Large BDP networks are known as **long fat networks (LFNs)** [73], and RFC 1323 [66] deal with these by specifying a byte shift count in a header option field, thus allowing `rwnd` to be the value of the window size field left shifted by the value in the shift count. A maximum value of 14 may be used for the shift count, allowing `rwnd` to be increased to a maximum value of $2^{16} \times 2^{14} = 2^{16+14} = 2^{30}$ bytes (≈ 1 GB). This is called TCP window scaling.



(a) Without Nagle's algorithm

(b) With Nagle's algorithm

Figure 2.9: TCP segment transmission time-line with and without Nagle's algorithm

2.5 Congestion control in TCP

While reliable data transfer is achieved through explicitly acknowledging received segments (packets), TCP also aims to be *robust* in the event of network congestion. TCP assumes packet loss in the network is due to increasing load and congestion building up. TCP therefore has mechanisms to adapt to this and limit the data rate to try to avoid this. The Internet relies on this adaptiveness, and is why the Internet protocol suite is referred to as TCP/IP.

2.5.1 Nagle's algorithm

Network congestion was identified as a potential serious problem as early as 1984 by John Nagle, and congestion collapse was described in RFC 896 [74]. One of the identified problems is what Nagle called the "small packet problem", where an application frequently delivers tiny chunks of data to be sent over TCP, e.g. an interactive Telnet [64] session that reacts on every single keystroke. This means that a host could, in a worst case scenario, transmit a 41-byte packet (segment header + 1 byte character) for 1 byte of useful data, and it would do so repeatedly.

Nagle proposed what is known as *Nagle's algorithm* to reduce the number of small segments sent, which is widely used by TCP implementations today [60]. The main idea is simple: buffer data chunks until they fill up an MSS-sized segment. Figure 2.9 shows how this works.

The algorithm is simple: whenever data is pushed from the application, buffer it until either the size of the segment reaches an MSS, or to a timer goes off. The latter ensures that data is ultimately sent if the applications stops pushing data. However, when a data segment is received, the current buffer can be

"flushed", as in sent as data payload with the ACK. This is because the ACK must be sent back anyway, so we can just allow the data buffered until now to piggy-back on that.

In the light of today's computers and networks, the algorithm has lost much of its relevance. Computers are vastly more powerful than at the time the algorithm was conceived. The network capacity is also much greater, and real-time applications relying on speedy transmission of data is becoming increasingly more common. Yet the algorithm is still enabled by default on many OSes. The double latency resulted by combining Nagle's algorithm with delayed acknowledgements, described in the next section, makes it terrible for interactive applications [4, 31].

2.5.2 Delayed acknowledgements

Let us revisit figure 2.7. Since ACKs are cumulative, it is unnecessary to send an ACK for every single segment when, strictly speaking, one ACK would suffice in the event that the entire window was received properly. In the case of of figure 2.7, an ACK for segment 3 (ack=4) and again for segment 7 (ack=8) would be enough.

This is exactly what RFC 1122 [60] recommends. Similar to Nagle's algorithm on sender-side, ACKs too can be delayed on the receiver-side to cover multiple segments. Two limitations are necessary, an ACK delay must not exceed 500 ms (the Linux kernel uses 200 ms [4]) and at least one ACK should be sent for every other segment received. The first requirement is self-explanatory, we do not want to trigger a retransmission as it would defeat the purpose. The latter requirement is needed because delaying ACKs disturbs RTT estimation. [4, 31]

If reordering is detected, then delayed ACKs must be disabled. Delayed ACKs is enabled by default in the Linux kernel [102].

2.5.3 Congestion window

The breakthrough in TCP congestion control came with the introduction of the **congestion window (`cwnd`)** in a paper by Van Jacobson in 1988 [10]. In section 2.4.4 we explained that TCP can control the sender rate by using a receiver advertised window (`rwnd`); while `rwnd` allows a TCP sender to react to the receiver's capacity, a mechanism to deal with the network capacity is needed. The solution was to introduce a *second* window, namely the `cwnd`. A TCP sender will always choose the minimum of the two windows when determining how much it should transmit. The size of `cwnd` is determined during TCP's four stages of congestion avoidance, which are described in RFC 5681 [72].

The specification differentiates between sender's MSS (SMSS) and receiver's MSS (RMSS), but in the following explanation we will keep it simple and use only MSS for describing the maximum size of a non-fragmented segment across the line. The specification also states that `cwnd` should be calculated as a number of bytes, many OSes (including Linux) uses number of MSS-sized segments instead of bytes [15].

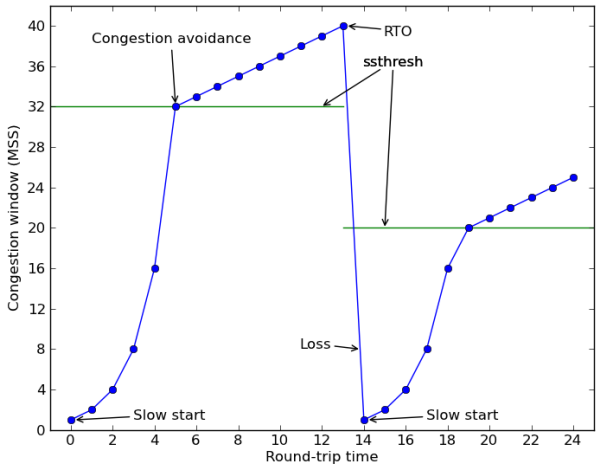


Figure 2.10: TCP slow start and congestion avoidance

Slow start

The first stage is used for determining the maximum available bandwidth the stream has. Although not slow at all, but slow compared to the original start-up stage in TCP, this stage is called *slow start* [30].

When the connection is established, the sender sets `cwnd` to the size of an MSS. If this segment is ACKed before the retransmission time-out (RTO) goes off, it adds another MSS to `cwnd`. If these 2 segments are ACKed successfully, the sender increases the window to $4 \times$ MSS and then again to $8 \times$ MSS and so on. In other words, the window size increases exponentially for each RTT as long as the segments are successfully ACKed. It does so until loss is detected or until the **slow start threshold value (`ssthresh`)** is reached.

Lost segments are interpreted as a sign of congestion building up in the network. If an RTO is triggered, `ssthresh` is set to `cwnd`/2, and `cwnd` is reset to its initial value. According to the specification [72], the initial `ssthresh` can be set arbitrarily high, but many implementations usually set it to 64 kBs [30, 31]. When `cwnd` exceeds `ssthresh`, the sender goes to the congestion avoidance stage.

Congestion avoidance

In the *congestion avoidance* stage, the window is incremented only by one MSS each RTT until loss is detected. On RTO, the sender sets `ssthresh` to `cwnd`/2, and goes back to slow start.

This additive increase on success and halving of `ssthresh` on loss, is called additive increase — multiplicative decrease (AIMD) and is essential to how the TCP transmission rate over time converges against the available capacity in the network. Figure 2.10 depicts how `cwnd` and `ssthresh` increases and decreases during these two stages and how they react to an RTO.

Fast retransmit

When a segment is lost from a transmitted window, the receiver will ACK the last received in-order segment. This can be used to infer that if the sender receives a **duplicate acknowledgement (dupACK)** for a sent window then the sent segment was not received properly (either out of order or lost).

The *fast retransmit* stage exploits this. If three dupACKs is received, the sender can be fairly certain that the segment is lost and not simply re-ordered by the network [31] [72]. Instead of waiting for an RTO to be triggered for the lost segment, the sender simply retransmits the segment "here and now".

RFC 5681 [72] states that during the fast retransmit stage, the sender must artificially inflate the window to compensate for the segments that triggered the three dupACKs. These data segments were in fact properly received, and are now in the receiver's buffer and not in the network. The specification states that `ssthresh` should be set to `cwnd`$/2$, and `cwnd` is then set to `ssthresh` $+ 3 \times$ MSS.

In addition, when performing the fast retransmit, a timer should be started. If the segment that led to the three dupACKs is recovered before this timer times out, a fast recovery is performed. Otherwise, the sender should go back to slow start.

Forward acknowledgement (FACK) is an extension to SACK that keeps track of the amount of data that has been received, and allows a sender to trigger a fast retransmit when it receives SACK information indicating loss of at least three segments [31,33]. It does this by assuming that all segments with a lower sequence number than the highest SACKed segment are lost.

Fast recovery

If a fast retransmit is triggered and the segment that led to the fast retransmit is recovered, the sender performs a *fast recover* and sets `cwnd` to the value of `ssthresh` and goes to congestion avoidance rather than going back to slow start. This is because the sender can assume the segment was lost due to sporadic loss in the network, and not due to congestion, since the subsequent segments was successfully received by the receiver. [72] [4, 30, 31]

Figure 2.11 shows how fast recovery works. The blue line depicts `cwnd` through the start-up (slow start) and into the congestion avoidance stage. At around the 8th RTT a fast retransmit is triggered, and the sender halves `ssthresh`. The blue line shows `cwnd` if the lost segment that lead to the fast retransmit is recovered (ACKed by the receiver before the fast retransmit time-out) and the sender goes to the congestion avoidance stage instead of going back to slow start. The red dotted line depicts what happens if the lost segment that triggered fast retransmit is not recovered, where the sender goes back to slow start.

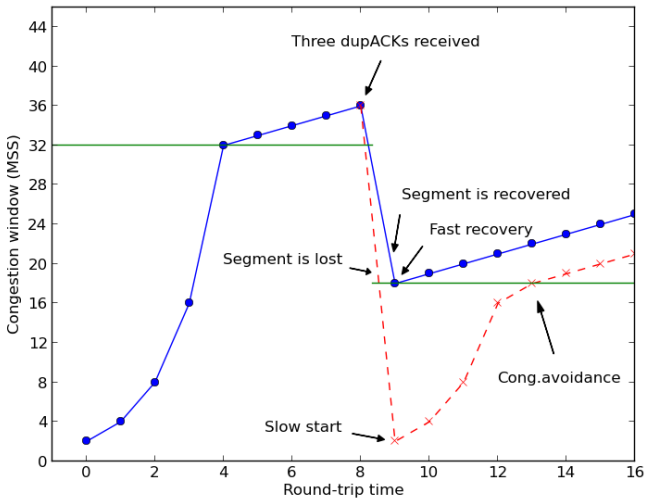


Figure 2.11: TCP fast recovery

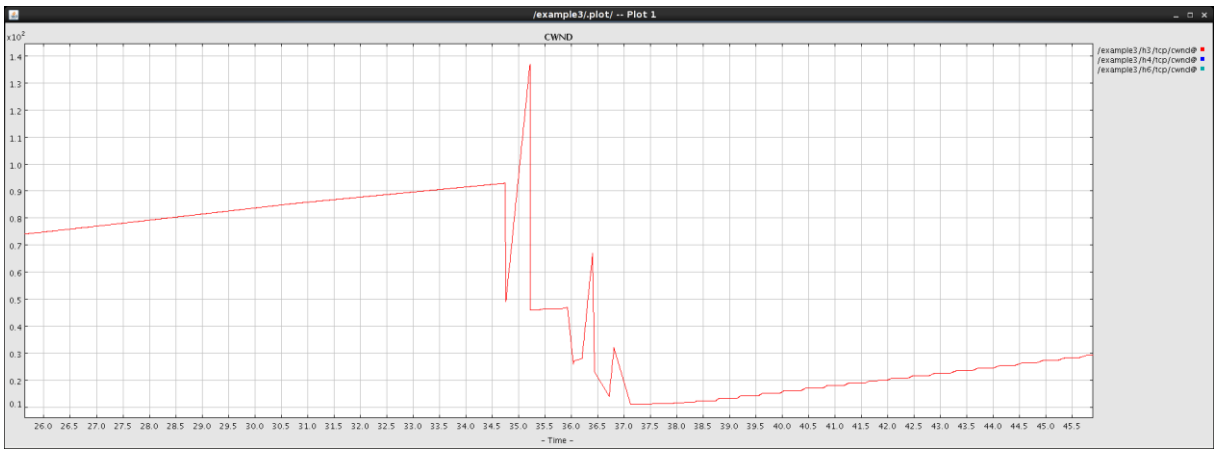


Figure 2.12: Fast recovery cycles in a J-Sim implementation of TCP Reno

Improved fast recovery

While traditional fast recovery is left upon receiving the first non-dupACK, multiple losses may still occur in the same window. A sender implementing a traditional fast retransmit + fast recovery will in that case leave and enter a fast retransmit + fast recovery cycle for each loss (and halve the window size each time!). Figure 2.12 shows a plot of `cwnd` for a TCP implementation in J-Sim using traditional fast recovery. The loss is caused by a bottlenecked router, so the lost segments are all from the same window. Notice how the fast retransmit and fast recovery cycle is entered three times.

RFC 6582 [75] proposes that the sender should stay in fast recovery until all the lost segments from the same window is acknowledged. To do so, the sender keeps track of the highest sent sequence number. When receiving three dupACKs and performing fast retransmit and fast recovery, the sender uses this stored sequence number to check whether the window is now accounted for or if there still are missing

segments. ACKs that cover the highest sent sequence number is considered a "full" ACK and mean that the window is recovered. ACKs that only account for some of the segments, are considered "partial" ACKs, and these are used for discovering "gaps" in the window. [4, 5, 31]

For example, imagine the scenario where a sender has transmitted segments 1, 2, 3, 4, 5 and 6, but 2 and 4 are lost along the way. The first "gap", the missing segment 2, is discovered when three dupACKs acknowledging only segment 1 are received (the receiver creates dupACKs upon receiving segment 3, 5 and 6). The sender then enters fast retransmit + fast recovery and retransmits 2, but stays in fast recovery. The receiver replies with an ACK accounting for 1-3 upon receiving segment 2 ("partial" ACK), thus allowing the sender to discover this new "gap", the missing segment 4, which is also retransmitted. After receiving segment 4, the receiver has now received the entire window, and sends an ACK acknowledging 1-6 ("full" ACK). TCP can now leave fast recovery, and `cwnd` is halved as usual. If a full ACK never arrives, an RTO will eventually occur and the sender will go to slow start.

However, this partial ACK mechanism is somewhat flawed. Without SACK, TCP is unable to distinguish between dupACKs caused by spurious retransmissions and dupACKs properly indicating loss. For example, if segments are reordered by the network by more than three segments, TCP mistakenly enters fast retransmit + fast recovery. But as the reordered segments are received, the acknowledgement number progresses and the sender then further mistakes these ACKs for partial ACKs indicating "gaps", and retransmits segments that are already received. [31]

2.5.4 Window algorithm variants

Although most TCP implementations incorporates the concepts and functionality we have described up to now, some variations exists. Today, there are many different variants for a variety of purposes. Examples include TCP Westwood+ [103], HighSpeed TCP [76], Compound TCP [104] and Data Center TCP [34]. The differences between the various implementations lie mostly in the congestion control mechanisms.

The first "modern" TCP implementation, was TCP Tahoe — a name given after the OS it was implemented in 4.3BSD-Tahoe. It came as a result of work with congestion collapse in ARPANET by Van Jacobson and the paper that followed [10]. TCP Reno (named after 4.3BSD-Reno) soon came after; the first implementation to support all four congestion control stages: *slow start*, *congestion avoidance*, *fast retransmit* and (traditional) *fast recovery* [4] [105]. We will in this section focus on the implementations most commonly used in the Linux kernel.

TCP NewReno

RFC 3782 [77] (updated by RFC 6582 [75]) defined an algorithm variant called NewReno. It uses the improved *fast recovery* stage instead of the traditional to detect multiple drops from the same window (see 2.5.3). NewReno was until recently the default TCP variation in many OSes, including the Linux kernel until version 2.6.8. In recent times, most implementations of Reno and NewReno also support SACK and DSACK in addition to forward RTO recovery (F-RTO) [78]. This reduces the problem with

indistinguishable dupACKs (explained in 2.5.3), but doesn't eliminate the problem entirely. This has been cited as one of the reasons it isn't the default congestion control in newer versions of the kernel [106]. The other reasons are related to poor scalability in networks with high BDP.

TCP Vegas

TCP Vegas [35] modifies Reno and adds fine-grained timers to TCP and tries to predict congestion based on variations in the RTT rather than packet loss. It also dynamically calculated the RTO, thus allowed quicker retransmissions of lost segments. However, the algorithm has a less aggressive *slow start*-stage, increasing `cwnd` every *other* RTT rather than every RTT, like e.g. Reno. This means that while it is effective in an all-Vegas environment [35], it is outperformed by more aggressive algorithms in terms of goodput when competing with these [107]. Vegas is implemented in the Linux kernel [108] and FreeBSD [109], and is the default congestion avoidance algorithm for the DD-WRT firmware (v24 SP2) [110].

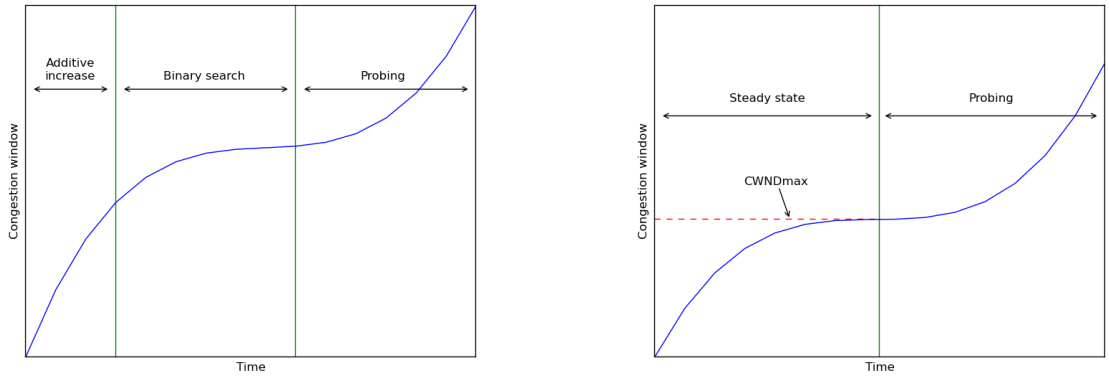
TCP CUBIC

The demands for fast transfer of large data volumes are ever increasing, and window scaling can only do so much. Binary increase congestion control (BIC) and its successor, CUBIC, are variants of TCP congestion control aimed at scaling for LFNs. [36] [111] [5]

The key concept of BIC is its window growth function: BIC maintains two variables, $\texttt{cwnd}_{min}$ and $\texttt{cwnd}_{max}$. Whenever packet loss occurs, $\texttt{cwnd}_{max}$ is set to the value of `cwnd`, `cwnd` is reduced by a multiplicative factor β and $\texttt{cwnd}_{min}$ is set to the value of `cwnd` after the reduction. Then BIC sets its `cwnd` to the midpoint between $\texttt{cwnd}_{min}$ and $\texttt{cwnd}_{max}$. This works because since packet loss occurred at $\texttt{cwnd}_{max}$, the window size the network can handle must be somewhere between $\texttt{cwnd}_{min}$ and $\texttt{cwnd}_{max}$. [36] [111]

To prevent a too aggressive growth, BIC defines a fixed constant S_{max}, so that if the difference between the midpoint and $\texttt{cwnd}_{min}$ is greater than S_{max}, BIC increments by S_{max} instead. If no packet loss occurs when incrementing `cwnd`, $\texttt{cwnd}_{min}$ is set to the value of `cwnd`. If the window incrementation is smaller than a fixed constant S_{min}, `cwnd` is set to the current maximum. These two threshold values ensure that after packet loss, the window growth function will resemble a linear function (*additive increase stage*) followed by a logarithmic function (*binary search stage*). When `cwnd` grows beyond $\texttt{cwnd}_{max}$, BIC enters a stage where it "probes" for the new maximum value by using a growth function symmetrical to the functions in additive increase and binary search — exponentially at first and then linearly. The three stages are illustrated in figure 2.13a.

To summarize the algorithm:



(a) TCP BIC

(b) TCP CUBIC

Figure 2.13: Congestion window growth stages

- Whenever cwnd is successfully increased and no loss occurs, $\texttt{cwnd}_{min}$ is updated and set to the value of cwnd ($\texttt{cwnd}_{min} \leftarrow \texttt{cwnd}$).
- Whenever loss is detected, $\texttt{cwnd}_{max}$ is updated and set to the current value of cwnd, cwnd is reduced by a multiplicative factor β and then $\texttt{cwnd}_{min}$ is set to the value of cwnd ($\texttt{cwnd}_{max} \leftarrow$ cwnd, then $\texttt{cwnd} \leftarrow \texttt{cwnd}/\beta$, then $\texttt{cwnd}_{min} \leftarrow \texttt{cwnd}$).
- For each RTT, cwnd is set to the midpoint between $\texttt{cwnd}_{min}$ and $\texttt{cwnd}_{max}$ ($\texttt{cwnd} \leftarrow \texttt{cwnd}_{min} + \lfloor(\texttt{cwnd}_{max} - \texttt{cwnd}_{min})/2\rfloor$). If cwnd grows beyond $\texttt{cwnd}_{max}$, BIC "probes" for a new maximum by updating $\texttt{cwnd}_{max}$ for every successful incrementation.

Even though BIC achieves relatively good fairness, its growth function was too aggressive in networks with short RTTs or with low bandwidth. In addition to this, the different growth stages added complexity to the algorithm. CUBIC is a simplified algorithm, which aims to improve the fairness when competing with other TCP implementations. It does so by replacing the stages in BIC with a single cubic function to calculate the window size. [111] [4, 5]

The cubic function that replaces the different stages in BIC can be expressed as:

$$\texttt{cwnd} \leftarrow C \times \left(T - \sqrt[3]{\frac{\texttt{cwnd}_{max} \times \beta}{C}}\right) + \texttt{cwnd}_{max} \tag{2.1}$$

where C is a scaling constant, and T is the elapsed time. As with BIC, β is the multiplicative decrease factor after a loss event. The authors of BIC and CUBIC argue that 0.4 is a good value for C, 160 is a good value for S_{max} and 0.2 is a good value for β. [111]

As illustrated in figure 2.13b, the window growth is similar to BIC, but CUBIC is much more "careful" around $\texttt{cwnd}_{max}$. The authors of CUBIC argue that using the elapsed time instead of relying on RTT yields a reasonable convergence speed and that CUBIC is more fair towards other streams than BIC. CUBIC also behaves better than other RTT-dependent algorithms in networks with a short RTT or low

BDP, but in order to be able to compete in these kind of networks, CUBIC — like many other high-speed TCP variants — has a fall-back mode which it uses if the window growth is too modest. [5] [111]

Until version 2.6.8, NewReno was the default TCP congestion control algorithm in the Linux kernel. BIC was the default from version 2.6.8 to 2.6.19, and CUBIC has been the default since. As of version 9 and onward, CUBIC is also available in FreeBSD [112, 113].

2.5.5 Retransmission time-out calculation

We mentioned in section 2.4.3 that TCP calculates a **RTO** for when segments should be retransmitted. This time-out interval relies on an estimation of the *network latency*, which TCP does by constantly measuring the **RTT** — the elapsed time between sending a byte with a certain sequence number and receiving an ACK acknowledging that specific byte.

More specifically, RFC 6298 [79] specifies that a TCP sender should keep track of the *smoothed RTT* (`SRTT`) and the *variance* (`RTTVAR`). The RTT should be measured at least once for each RTT[5], and *Karn's algorithm* must be used when taking RTT measurements. Karn's algorithm states that RTT measurement from retransmitted segments must be ignored because it is ambiguous whether the ACK acknowledges the segment from the previous transmission (the ACK is simply delayed) or if it acknowledges the current retransmission.

Initially, the RTO should be set to 1 second. After the first RTT measurement, a host should set its `SRTT` to this value, while `RTTVAR` should be set to $\frac{\texttt{SRTT}}{2}$. Ignoring the clock granularity, whenever a new RTT measurement RTT' subsequently is made, the host should compute the following

$$\texttt{RTTVAR} \leftarrow (1-\beta) \times \texttt{RTTVAR} + \beta \times |\texttt{SRTT} - RTT'| \tag{2.2}$$

$$\texttt{SRTT} \leftarrow (1-\alpha) \times \texttt{SRTT} + \alpha \times RTT' \tag{2.3}$$

where $\alpha = \frac{1}{8}$ and $\beta = \frac{1}{4}$ according to the specification [79]. α says how strong new measurements should influence the `SRTT`, and β determines how quick RTOs should be triggered [31].

The RTO should then be calculated as

$$RTO \leftarrow \texttt{SRTT} + 4 \times \texttt{RTTVAR} \tag{2.4}$$

If RTO is less than 1 second, it should be rounded up to 1 second [79], but some OSes (like Linux) have a lower value for this RTO_{min} to avoid unnecessary high retransmission delays. However, choosing a very low RTO_{min} may trigger retransmission even though the segment is received and an ACK is under way. If achieving a higher throughput is the goal, this is a waste of resources [4, 5].

Whenever a RTO goes off and a segment is retransmitted, RFC 6298 [79] mandates that the RTO interval must be doubled for each successive retransmission of a segment until it is acknowledged. This is known

[5] RFC 1323 [66] suggests that TCP connections that uses large windows should take several RTT measurements per window to avoid aliasing effects in the estimated RTT.

as **exponential back-off** and is a mechanism to prevent congestion collapse should severe congestion build up in the network, allowing streams to entirely withdraw from trying to transmit.

2.6 Thin-stream modifications to TCP

Due to TCP delivering data in-order, when a segment is lost and has to be retransmitted, the receiving application is blocked until the lost segment is successfully delivered and TCP can push that and subsequent segments to the application — even if all of the subsequent segments are received properly. This is often referred to as *HOL blocking*. Greedy streams are able to compensate for the delay introduced by waiting for an RTO by triggering retransmission already on three dupACKs, as explained in section 2.5.3. SACK and FACK improves this further, by supplying the sender with more detailed information about lost segments so that fast retransmit can be triggered faster [6, 31].

Thin streams are not as lucky. They have a high inter-transmission time (ITT) between data segments (relative to the RTT), which means that they are prone to have very few packets in flight (PIFs). When the number of PIF is lower than four, something that is very likely for most of the application-limited streams shown in table 2.1, a serious problem occurs: there simply are not enough transmitted segments to trigger a fast retransmit + fast recovery cycle if the window is lost. The implication of this is that not only do thin streams react much later to congestion than greedy streams, but they also suffer horrible latencies caused by retransmission delays. [4–6, 8, 15]

Other types of application-limited streams are also affected by the three dupACKs limit for triggering fast retransmit. TCP streams such as bursty and short-lived[6] HTTP streams are vulnerable to *tail loss* — dropped segments at the end of the stream. This can have a serious impact on the stream's completion time [80,81] [8,38,39]. Because the lost segments are among the last segments, the number of subsequent segments are not enough to trigger three dupACKs and the lost segments can only be recovered after a time-out.

Analysis of Google web server traffic show that almost 70% of retransmissions for short-lived HTTP connections are triggered by an RTO and only 24% are recovered by fast recovery, while longer-lived HTTP connections streaming YouTube video, RTO triggered retransmits account for 46% of the retransmissions [7].

2.6.1 TCP smart framing

A TCP stream sending fewer than seven MSS-sized segments, i.e. 10 kBs spread out over 1460-byte-sized segments, has no chance of triggering a fast retransmit. To make the matter even worse, if the receiver uses delayed ACKs as explained in section 2.5.2, a stream sending less than ten segments will never trigger a fast retransmit.

[6] Short-lived flows usually transmit only a handful of segments, with average lengths no longer than 10 kB [7, 8, 37]

A modification attempting to reduce the flow completion time for short-lived streams is TCP smart framing (TCP-SF) [8] [114]. In order to increase the number of ACKs, the full-sized segments are divided into smaller sized segments. Not only does this increase the chance of getting three dupACKs, but it also increases the accuracy of the RTT sampling, which leads to an improved RTO calculation. This means that the penalty of the high initial RTO value for networks with a short RTT is alleviated.

It is argued that as long as the stream short-lived and transmits little data, the added load of sending more but smaller segments compared to sending fewer but larger segments is minimal [8]. This means that Nagle's algorithm, explained in section 2.5.1, must be disabled when using TCP-SF.

TCP-SF operates with an *alternative* MSS, which can be expressed as

$$\text{MSS}_{alt} \leftarrow \frac{\texttt{cwnd}_{init}}{\texttt{cwnd}_{thresh}} \tag{2.5}$$

where $\texttt{cwnd}_{init}$ is the initial `cwnd` size, and $\texttt{cwnd}_{thresh}$ is a threshold value. The threshold value determines when to use the alternative MSS and when to use the default MSS. In other words, if $\texttt{cwnd} < \texttt{cwnd}_{thresh}$ then MSS_{alt} is used; otherwise the default MSS is used. The authors suggest that because triggering a fast retransmit requires three dupACKs, $\texttt{cwnd}_{thresh}$ should be 4 [8]. This corresponds to the PIF limitation mentioned above.

For each in-sequence ACK received, the MSS should be increased with a factor α. The authors suggest using $\alpha = 1.32$, but it can also be 0 for simple implementations which means that a fixed minimum MSS is used [8] [114]. As soon as `cwnd` reaches the threshold, $\texttt{cwnd}_{thresh} = 4$, the default MSS should be used.

TCP-SF is not implemented in the Linux kernel. The modification is beneficial when large segments, i.e. MSS-sized segments, are sent as smaller-sized segments. However, as seen in table 2.1, thin streams often send small packets, which means that there is not a lot of data to divide into smaller segments.

2.6.2 Early retransmit

RFC 5827 [80] addresses the limitations surrounding PIF and the threshold of three dupACKs for triggering a fast retransmit. A mechanism called early retransmit (ER) is proposed, which allows the TCP sender to require fewer dupACKs before it performs a fast retransmit in some circumstances.

The mechanism is quite conservative, only allowing the number of required dupACKs to be reduced if the amount of outstanding data is less than four segments and if the sender is otherwise inhibited from sending new data — either that the advertised `rwnd` does not permit new data to be sent, or that there is no new data ready in the sender's transmit buffer.

However, as explained in 2.5.3, the reasoning behind the choice for requiring three dupACKs before triggering fast retransmit, is to be fairly certain that the dupACKs are caused by a packet loss and not network re-ordering [31]. This means that ER is prone to creating spurious retransmissions for application-limited streams in networks where packet re-ordering is prevalent, but is somewhat more reliable when used in

conjunction with SACK enabled and delayed ACKs disabled [80]. Even so, the RFC recommends disabling ER in case the stream has detected re-ordering. Studies show that adding a small delay to early retransmissions is effective in preventing spurious retransmissions due to re-ordering because it gives the ACKs time to arrive and cancel the pending retransmit [7].

ER was implemented in the Linux kernel in version 3.5 [115] and is enabled by default [101].

2.6.3 Modified fast retransmit

Another improvement dealing with the problem of number of dupACKs contra the number of PIF, is MFR. Similarly to ER, it aims to reduce the number of dupACK required to trigger a fast retransmit. Unlike ER however — where both the `cwnd` and outstanding bytes not yet transmitted determines the number of dupACKs required to trigger a fast retransmit — MFR enters fast retransmit already on the first received dupACK (as long as the stream has fewer than four PIF) [4].

This *faster* fast retransmit is beneficial since the sender avoids going back to slow start and avoids triggering exponential back-off. Although going back to slow start or not has little effect on a thin stream, as long as the implementation calculates the initial `cwnd` as a factor of MSSes [5], avoiding having to wait for an RTO and avoiding increasing the RTO time (exponentially) is hugely beneficial in terms of reducing application latency introduced by retransmission delays. It is argued that the lowered retransmission delay justifies the risk of potentially creating unnecessary retransmissions, and since the thin stream does not expand its `cwnd` during recovery, it will not contribute to renewed congestion. [4–6].

The MFR algorithm has been part of the Linux kernel since version 2.6.34 [9] [116].

2.6.4 Linear retransmission time-out

Retransmissions are the most crucial factor for networking delay for thin streams. Experiments show that the most extreme latency events were when a data segment had to be retransmitted several times [4, 5]. The reason for this is the exponential back-off algorithm for calculating the RTO timer: for each time the segment has to be retransmitted due to an RTO, the RTO grows by a power of two. To make matters worse, successive RTO events can also be caused by delayed ACKs (see 2.5.2), and not only loss events, especially if the IAT is high.

Disabling exponential back-off for thin streams and instead increasing the time-out linearly, was proposed as an enhancement in order to compensate for the extreme latencies experienced for consecutive RTOs [4–6]. It was included in the Linux kernel for version 2.6.34 [9] [117]. In order to avoid being too aggressive on retransmission, the implementation in the kernel will only use linear back-off for six consecutive RTO events, before it starts increasing the time-out exponentially.

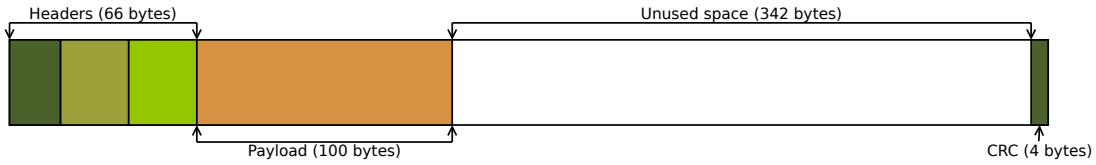


Figure 2.14: Unused space in an Gigabit Ethernet frame when transporting a 100 byte payload

2.6.5 Tail loss probe

While ER and MFR reduces the number of dupACKs required to trigger a fast retransmit + fast recovery cycle, studies show that as much as 96% of RTOs for short-lived HTTP streams occurred without a single dupACK in advance [7] [118]. Tail loss probe (TLP) [118] attempts to solve this by "probing" the receiver with a segment containing unacknowledged data if the sender has not received any ACKs for a certain amount of time. If tail loss has occurred, the ACK for the loss probe will trigger a SACK- or FACK-based fast retransmit. This means that TLP only works with SACK enabled.

TLP was implemented in the Linux kernel in version 3.10 [119], and is enabled by default [101].

2.6.6 Redundant data bundling

The thin stream modifications mentioned so far deal with the problem of having too few PIF to trigger a fast retransmit. As we can see in table 2.1, segments belonging to a thin stream do not only have a high IAT, but they are also on average smaller than an MSS. In addition, Gigabit Ethernet uses a minimum frame size of 512 byte (4096 bit time slots) regardless of what is actually being used [40], as figure 2.14 illustrates. In other words, a lot of space is left unused in every transmitted segment. Redundant data bundling (RDB) is a modification that attempts to exploit this, by bundling previously sent but not acknowledged data in every new segment. [4, 5, 15]

Figure 2.15 demonstrates how this work. The application is sending messages of 365 bytes at a constant interval. The first segment is lost, so on the second write the data (grey) from the first segment is bundled in the new segment (olive). However, the second segment also gets lost, so the third time the application does a write data from both the first (grey) and the second segment (olive) is bundled with the new data (purple). The third segment arrives, carrying the continuous data, so the cumulative ACK covers all three segments. The sender then transmits a fourth segment with new data only (brown). This time, the ACK is lost. The fifth time the application does a write, the data from the fourth segment (brown) is bundled with the new data (green).

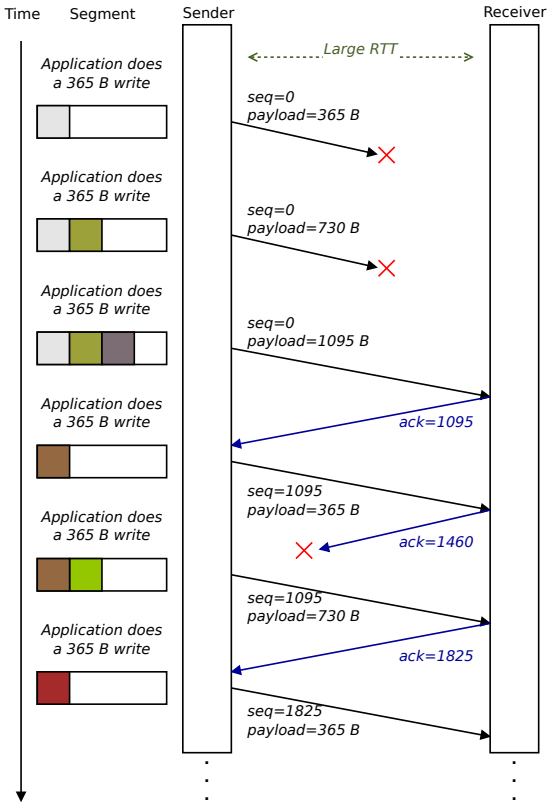


Figure 2.15: Redundant data bundling

2.7 TCP fairness

In addition to preventing congestion collapse, the underlaying principle of the TCP congestion avoidance algorithm explained in section 2.5.3 is fairness between TCP streams. The idea is that adapting the transmission rate according to the state of the network, i.e. how saturated it is, ultimately allows the capacity of the network to be distributed fairly among multiple TCP streams.

However, simply dividing the bandwidth evenly between streams is not very efficient. Consider the example network scenario shown in figure 2.16. There are three routers with two shared links between

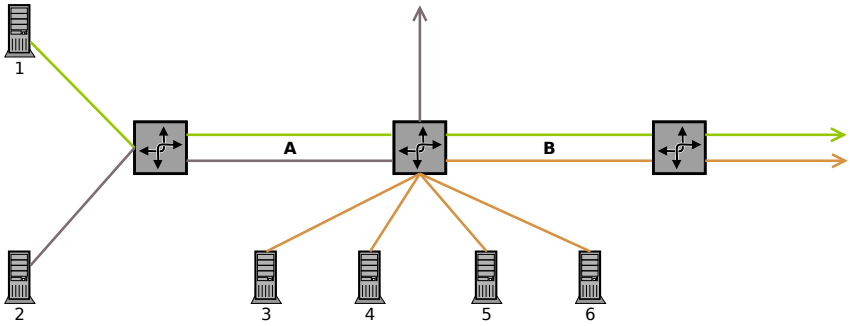


Figure 2.16: Example network with six senders and two shared links

them, A and B. A and B have a total capacity, C, each. Host 1 and 2 are connected to the first router, while host 3–6 are connected to the second router. All hosts have a fixed throughput, x_i, for their respective connections. Note that host 2 only uses the shared link A, while hosts 3–6 only use the shared link B. Host 1 is the only host with a connection that uses both shared links.

With an equal bandwidth share as our fairness criterion, we would end up with a throughput $x = 1/6 \times C$ for all streams. This would be a fair division of the total capacity, but it would also be a poor link utilization of the shared link A. Only $\frac{2}{6}$ of its total capacity would be in use.

2.7.1 Max-min fairness

The idea of max-min is that bandwidth share (throughput) of the smallest streams should be as large as possible. Given this condition, the bandwidth share of the second smallest streams should be as large as possible, and so on. Each stream's bandwidth share only stops growing when one or more links on the path reaches its maximum capacity [13] [49] [120].

Max-min fairness can be said to be achieved when an attempt to increase the share by any stream results in the decrease of the share for some other stream with an equal or smaller share. More formally, let $\vec{x}$ be the distribution of received shares for n streams, $x_1, x_2, \ldots, x_n$. If the distribution is max-min fair, then for any alternative distribution of shares, $\vec{x'}$, if $x'_i > x_i$, there must exist some j so that $x_j \leq x_i$ and $x'_j < x_j$.

Lets apply a progressive filling algorithm on our example scenario depicted in figure 2.16. All streams start with a bandwidth share of 0, and their sending rate grow at the same pace. At some point, the streams 1 and 3–6 will hit the limit of link B. At this point all streams have a bandwidth share $x = 1/6 \times C$. Any attempt by stream 1 or 3–6 to increase their send rate, will result in a decrease for some of the other streams, so they stop increasing. Stream 2, however, is still able to increase its share without affecting the other streams. It will continue to increase its bandwidth share until the link capacity of link A is reached, because at that point any attempt to increase the rate will result in a decrease of the share of the other streams. This means that when the max-min fairness is achieved, the bandwidth share for stream 1 and 2 will be $x_1 = \frac{1}{6}C$ and $x_2 = \frac{5}{6}C$ respectively, and both shared links are fully utilized.

Max-min fairness through progressive filling is the key element of the TCP congestion avoidance algorithm presented in section 2.5.3 [10, 12, 13, 15] [120]. If n TCP streams share the same link, they should all get an equal share, $1/n \times C$, where C is the total capacity of that link. Assuming that loss is a result of link saturation, something most TCP congestion window algorithms do, this means that a TCP stream that experiences packet loss does so because it has exceeded its fair share of the bandwidth. The idea is that by backing off on loss events, the streams will eventually converge against an equilibrium. [10,13,15]

This is what is known as the **TCP fairness principle**. However, TCP is not true to the progressive filling algorithm; when several TCP streams with different RTTs share a link, they do not increase their `cwnd` at the same pace. In other words, the streams with longer RTTs will be discriminated against

because the streams with a short RTT will be able to increase their `cwnd` faster during slow start and loss recovery [5, 12, 15].

How well a transport protocol manages to be fair when competing against TCP streams over a shared resource is what is referred to as *TCP friendliness* [15, 31] [111]. This term is most often used when describing how non-elastic streams affect TCP streams.

2.7.2 Jain's fairness index

Jain's fairness index [11] is often used to evaluate fairness. It states that if n streams share a resource (i.e. the same limited link), and $x_1, x_2, \ldots, x_n$ are the received shares of that resource for those streams (i.e. the throughput of those streams), then the fairness index, J, is expressed as

$$J(x_1, x_2, ..., x_n) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} {x_i}^2} \tag{2.6}$$

where x_i is the received share of the resource for the i-th stream.

Jain's fairness index ranges from 0 to 1, and is 1 when all streams receive the same share. It is k/n when k streams equally share the resource, and the remaining $n - k$ streams receive a zero-sized share of the resource.

When dealing with streams that have different bandwidth requirements, we have to use the throughput relative to the optimised throughput. Picture a scenario where we have two streams sharing a 1 Mbps link; a thin-stream sender transmitting 120 bytes every 100 ms and a greedy-stream sender trying to consume the rest of the bandwidth not used by the thin-stream sender. Even when the thin-stream sender receives the optimal amount of bandwidth, $\frac{120 \times 8}{0.100} = 9600$ bps, or 9.6 kbps, using simply the throughput alone will lead to an index considered only 51% fair:

$$J(9.6, 990.4) = \frac{(9.6 + 990.4)^2}{2(9.6^2 + 990.4^2)} \approx 0.51$$

Because of this we adapt the index to use the throughput, x_i relative to the optimal throughput, o_i, instead:

$$J(\frac{9.6}{9.6}, \frac{990.4}{990.4}) = \frac{(1+1)^2}{2(1^2 + 1^2)} = \frac{2^2}{2(2)} = 1$$

We should always carefully consider the proper criteria for the optimal situation when using Jain's fairness index.

2.8 Summary

The congestion avoidance algorithm in TCP uses packet loss as an indication of the network capacity. By maintaining a window called `cwnd`, which is increased by actively probing for more bandwidth and

decreased when loss is experienced, TCP adapts its transmission rate according to the available capacity. This algorithm, however, assumes that TCP streams tries to consume as much bandwidth as possible.

Thin streams are a type of streams with a different traffic pattern: small packets sent at a low rate. These are often created by applications that have latency requirements rather than bandwidth requirements. Whereas greedy TCP streams are limited by `cwnd`, thin streams do not probe the network for available capacity and are limited by retransmission mechanisms instead. Modifications to these mechanisms, such as MFR and LT, aims to reduce the latency the unmodified mechanisms introduce by retransmitting more aggressively.

The effects of the TCP congestion control have been studied for decades, while the impact of thin streams is relatively unexplored. In the next chapter, we discuss how we design an experiment to test and evaluate the thin stream modifications available in the Linux kernel in order to assess a reasonable trade-off between reduced latency for thin streams and how an increased aggressiveness affects the system.

Chapter 3

Experiment design and tools

Thin streams are without any doubts suffering from mechanisms designed for limiting streams that actively try to use as much of the available bandwidth as possible. The modifications presented in section 2.6 manage to reduce the penalty these streams experience by somewhat reducing the latency cost for retransmissions. It has been suggested that these modifications are justifiable, because thin streams do not increase their `cwnd` during recovery and thus will not contribute to renewed congestion [4]. However, even with modern techniques for traffic shaping and rate limiting, many networks are still at the mercy of elastic streams, such as TCP, and are completely dependent on their mechanisms for avoiding congestion in order to prevent congestion collapse. Making TCP more aggressive is therefore something that should be done with great care.

With the exception of some preliminary tests, testing and analysing the general TCP friendliness of these modifications is something that until now has remained relatively unexplored. Despite this, two thin-stream enhancements in particular, MFR and LT, are already included in the mainline Linux kernel branch as of version 2.6.34 although not turned on by default [116, 117]. In this chapter we discuss the design and implementation of a fairness experiment in an attempt to assess the fairness of these modifications towards other streams.

3.1 Metrics

To be able to conduct our experiment, we must first define what we consider to be fair and how we can measure this fairness. Traditionally, fairness has been defined as streams having an equal share of the bandwidth of a shared link [10,11]. More recent work, however, use a variety of criteria when judging the fairness of a stream's behaviour and how it affects other streams [12] [49]. Using bandwidth allocation as a measurement of fairness has been heavily criticised [12], but is nevertheless still widely used today.

Our motivation for running a fairness experiment is to examine how the thin-stream modifications to the congestion avoidance algorithm affect other TCP streams. Based on the observation that time-dependent

applications often generate thin streams (see section 2.1), as well as the recommendations in RFC 5166 [49], we have selected three metrics in particular to test our hypotheses stated in section 1.4.

The first metric is bandwidth share allocation. According to the TCP fairness principle, all TCP streams should receive a fair share of the total available link capacity. Although this is not true when streams with different properties compete, as explained in section 2.7 and 2.7.2, we can still use the throughput to compare the share the streams get with what they should have received had the conditions been optimal.

The second metric is latency. The purpose of the thin-stream modifications is first and foremost to improve the application-experienced latency. Therefore, we must make sure that improving the latency of thin streams using these modifications does not increase the latency experienced by the thin streams using unmodified retransmission mechanisms. In addition, (increasing) latency, and queuing delay in particular, is an indication of congestion building up in the network, so we will also consider delay variation in our analysis of latency.

The third metric is loss. Loss is used by most TCP implementations as indication of congestion (see 2.5 and 2.7). Moreover, the goal of the thin-stream modifications is to recover from loss faster, since the retransmission delay has been shown to be a major factor contributing to the overall latency [4, 5]. It is therefore imperative that having more aggressive retransmission mechanisms does not cause other streams to experience increased loss.

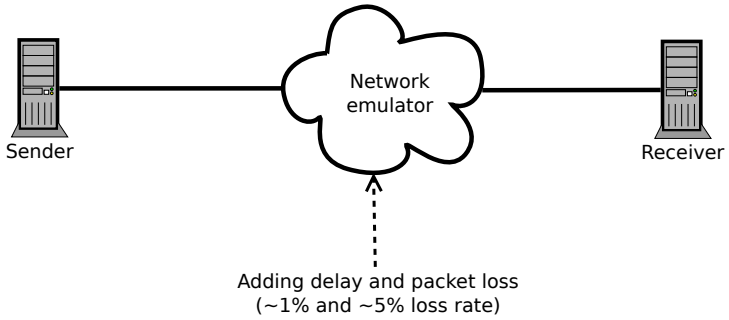
3.2 Design considerations

In order to conclusively determine whether our hypotheses hold up or not, we need to design our experiment to model the real world as much as possible. At the same time, in order for our analysis to be correct, there is an inherent desire for the test environment to be as deterministic as possible. In this section we will discuss some considerations regarding how to best design the experiment as well as mentioning some concerns regarding the limitations of the previous evaluations of the effectiveness of the thin-stream retransmission mechanisms.

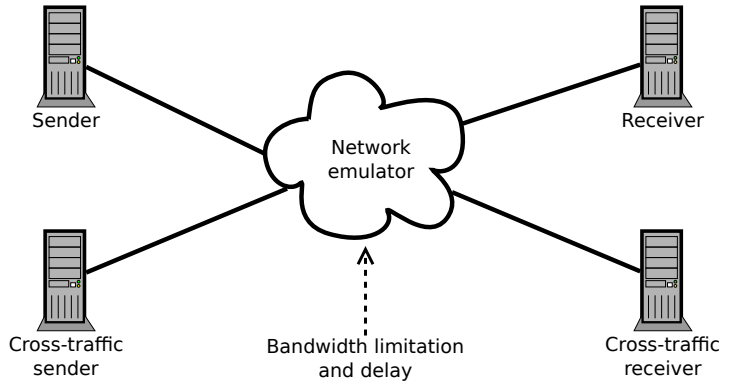
3.2.1 Previous evaluations

The previous evaluations of the thin-stream modifications MFR and LT have mainly been focused on showing their effectiveness in reducing the application-experienced latency. In order to do so, the authors of these studies induced a certain amount of loss to trigger dupACKs and RTO events for MFR and LT respectively [4, 5]. We will now briefly revisit the test scenarios and set-up used in these evaluations.

Figure 3.1a shows one of the network emulation scenarios used in those evaluations: a sender host establishes connections and transmits data to a receiver host through a network emulator (`netem`) [121]. The network emulator added delay to simulate an RTT, jitter and dropped packets randomly from a uniform distribution to ensure that enough loss events occurred, as well as occurrences of 2nd through



(a) Network emulator creates loss and delay



(b) Loss is caused by congestion

Figure 3.1: Network configurations used in previous evaluations

n-th retransmissions in order to test the efficiency of LT. However, a test where a thin stream sender with an IAT of 100 milliseconds and the emulator has a drop rate of 1%, a test would have to run for ten hours in order to generate 3600 loss events.

A second emulation scenario used, was to test the modifications in a more realistic loss scenario. As depicted in figure 3.1b, traffic was still sent from a sender to a receiver. However, unlike the first scenario, the network emulator did not drop packets. Instead it was configured to limit the bandwidth on the outgoing link. Two extra machines was used in order to generate cross traffic, mimicking traffic similar to HTTP. The IATs of the thin-streams was chosen from a Pareto distribution with a chosen mean value, and the length of the HTTP-like streams was varied. The length of the queue in front of the bottleneck was set experimentally — a sensible trade-off between queueing delays and loss was chosen.

For generating foreground traffic, the authors used two different methods. One of the methods was developing a tool called `streamzero` (see 3.3.5) that simulates a thin-stream application by periodically trying to transmit small amounts of data. This generates small TCP segments that is sent out on the wire periodically, similar to the traffic patterns discussed in section 2.1. The other method used in the previous evaluations, was capturing real-life thin-stream traffic from various interactive applications [4], as shown in table 2.1, and creating a self-made program to replay this traffic [6].

A third test scenario the authors of the previous studies used, was sending thin streams through the Internet. Different Internet Service Providers (ISPs) were used for the experiment in order to test different

network characteristics and path conditions. In addition to evaluating the effects of the thin-stream modifications in a real world scenario, it also allowed for a user evaluation of perceived latency by using the modifications on traffic generated by real interactive applications [4, 15].

Another related study attempted to create a model of different thin streams in order to classify them [13]. This study did not test the thin-stream modifications, but rather evaluated how thin streams in general behave. A network simulator called ns-3 [122] was used as the test environment for these tests, and the Linux kernel implementation of TCP was used in the simulations. However, the author of this study observed unexpected effects and was unable to conclude whether these were real network effects or a result of the simulation itself. Synchronisation caused by simulator ticks was pointed out as candidate explanation.

3.2.2 Simulation versus emulation

A method often used in networking research, is using a network simulator. These simulators are models of reality, i.e. real computer networks, and can therefore be used to evaluate and investigate various properties and mechanisms used by protocols and network components. A commonly used network simulator, is ns-2 [123] [14]. The thin-stream modifications we want to evaluate has not to our knowledge been implemented in ns-2. In addition, the ns-2 TCP implementation differs from the Linux TCP implementation on other areas as well [41, 42].

Even though the modifications we want to test are not implemented in any network simulator known to us, it is still possible to work around this. One of the improvements of ns-3 over ns-2, is the Direct Code Execution (DCE) framework [13, 14] [124]. DCE allows user-space and kernel space code to be executed as part of the simulator, meaning that the Linux networking stack can be used in simulations. This level of abstraction is quite resource intensive and can, according to one of the previous studies discussed in the section 3.2.1, lead to uncertainties surrounding the implementation of the simulator. Since we aim to determine the right level of aggressiveness that can be justified for a thin stream, it is *critical* that we understand our testbed and eliminate such uncertainties. A study suggest that different simulators can give wildly different results for the same test scenarios [14].

A second possibility is to send traffic between two hosts over the Internet. By varying the ISPs the sender and receiver are connected to, the effects can be measured with different network conditions depending on the path and competing traffic. However, the downside of these tests is that the observer has no influence over the network properties. While this approach is useful for testing the effects of the thin-stream modifications in a realistic setting, as seen in the previous evaluations, the lack of determinism and control makes it challenging for a fairness experiment.

The third way, is conducting evaluations by emulation. Emulation differs from simulation in that a simulation attempts to model reality while an emulation uses real components mixed with emulated aspects. In other words, we can replace a real-world component, such as a large network, with an emulator. This allows us to test the actual implementation of the thin-stream modifications in a setting

close to reality, while still retaining a controlled environment in which we influence all conditions and parameters.

We have chosen emulation as our approach for our evaluations. The main reason for this is that we want to avoid any uncertainties surrounding network simulators, as experienced in the previous thin stream study mentioned in the previous section [13]. In addition, we argue that since the behaviour of thin stream traffic is still something that is not well-investigated, using real-life network components allow us to identify any sources of error that stem from how hardware or the OS treat thin stream network traffic.

3.2.3 Realistic packet loss

Using a drop scheme that randomly drops packets from a uniform distribution is suitable for ensuring that a statistical significant number of loss events occur. However, it does not necessarily reflect real-world circumstances that well. As explained in section 2.5, TCP congestion control assumes that loss is a result of network congestion. Congestion occurs when the contention for a shared resource exceeds the capacity of that resource. The implications of this is that a realistic method of generating loss events is to introduce competition over a shared resource.

In a packet-switched, statistically multiplexed network, like IP networks are, routers use *queues* to buffer packets. Since a router's memory is finite, these queues are also finite. When a router gets saturated, it will stop buffering packets and drop them instead. This is normally what happens when a router's outgoing network link is congested; packets are received at the router at a higher rate than the router manages to forward them. The router's queue begins to grow, and if the high send rate continues, the queue will ultimately become full.

How routers decide what to enqueue and what to drop is an entire field of study in itself, but despite years of research and a lot of efforts put into improving these drop schemes and active queue management (AQM) techniques, the most common scheme actually deployed today is a tail-dropping first in — first out (FIFO) policy [2, 43, 44]. This means that when the router's forwarding queue is full, it will simply drop incoming packets until the queue starts to clear. Intuitively, we see that when loss is caused by a saturated router, the probability of packet loss is determined by the drop scheme used by that router and is not necessarily uniform.

3.2.4 Generating thin streams

We saw in section 3.2.1, that in order to evaluate the effectiveness of the thin-stream retransmission modifications, one of the methods the authors used was to capture real network traffic from some thin-stream applications and create a tool called `tracepump` to replay the data packets of that traffic [4–6, 15]. While this is a suitable approach for generating traffic with realistic packet sizes and ITTs[1], it does however have some potential pitfalls one needs to be aware of.

[1] Please note the difference between IAT and ITT; the former is suitable when evaluating packet traces, while the latter is used when describing sending data at predetermined intervals.

Our main concern with `tracepump` is that it attempts to emulate a thin-stream application with limited data. By using only the packet traces means that the only data available is what is actually sent on the wire. Information such as which socket options were used, internal buffer sizes, congestion control mechanisms and the system's load is lost. In addition, some networking factors such as the RTT and how congested the network is, can also be difficult to determine. Any relationship or correlation between ITT and RTT or packet size and RTT, as well as the effects of network congestion or link stability, is hard to determine outside a controlled environment with a single trace.

The second method used by the authors of the thin-stream modifications in their evaluations, was to analyse the captured traffic (as seen in table 2.1) and use another program, `streamzero`, to simulate such traffic using realistic packet sizes and ITTs, as well as packet size variance, gathered from their analysis [4,15]. The strength of this approach, unlike the other approach mentioned above, is that the data gathering can be repeated and done under different circumstances, e.g. multiple servers and clients from different locations, thus eliminating uncertainties and doubts surrounding the significance of the data when the data is gathered from an uncontrolled environment. The weakness of this approach however, is that in order to create a correct model of the network traffic generated by a thin-stream application, we must have an understanding of how the thin-stream application is supposed to work. In addition, making the methods for analysis precise is difficult because of the huge amount of factors that might influence the data.

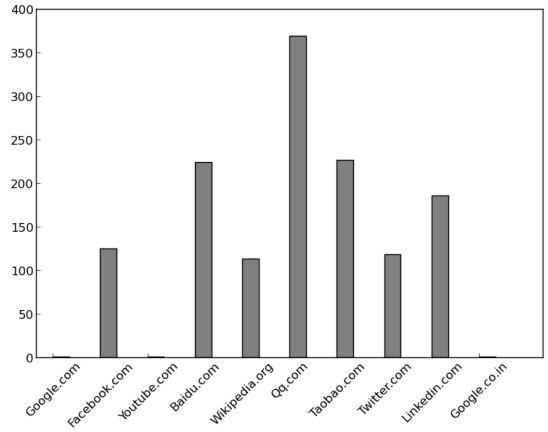
3.2.5 Choosing the network parameters

We still want a deterministic test environment in order to determine the fairness of the modifications and avoid having our tests influenced by factors outside our control, yet we also wanted realistic network conditions and parameters. Modelling reality for an experiment, however, demands a consideration of various network conditions such as packet loss, packet re-ordering and jitter. By conducting a simple test, we attempted to establish how common these are in the Internet.

Our initial idea was to establish HTTP connections to a number of web-sites and gather statistics from these flows using a packet analyser. However, modern day web-sites use multiple web servers and put these behind content delivery networks (CDNs), firewalls and application-layer gateways (ALGs), load-balancers and reverse proxies [7], which can lead to packet re-ordering and varying RTTs that occurs at the end-point [39,45]. This is not what we wanted to measure — we were interested in the properties of the networks along the path.

Instead, we decided to do a much simpler test using the `ping` utility program. This utility sends an Internet Control Message Protocol (ICMP) [82] echo request with a sequence number to a host and measures the number of milliseconds until it gets an echo reply with the same sequence number in return. This means that we only measured network latency.

Table 3.1 shows a subset of the results of measuring the RTT to the top 100 most popular web-sites from



		Round-trip delay time (ms)			
		min	avg	med	max
Google.com	81.175.29.177	0.8	0.9	0.9	2.7
Facebook.com	173.252.110.27	125.0	125.4	125.0	128.0
Youtube.com	81.175.29.148	0.8	0.9	0.9	1.4
Baidu.com	220.181.111.86	224.0	224.0	224.0	225.0
Wikipedia.org	208.80.154.224	113.0	113.0	113.0	118.0
Qq.com	125.39.240.113	361.0	368.7	368.0	374.0
Taobao.com	42.120.194.11	227.0	227.0	227.0	227.0
Twitter.com	199.16.156.230	118.0	118.0	118.0	119.0
Linkedin.com	216.52.242.86	185.0	185.5	185.0	187.0
Google.co.in	81.175.29.154	0.8	0.9	0.9	1.0

Table 3.1: RTT from Simula to the ten most popular web-sites in June 2014

the Alexa web-site ranking[2]. The measurements was taken from Simula Research Laboratory using their Internet uplink. 1000 echo requests was sent in total to each host in bursts of 20 at the time using a 10 millisecond interval. Sending ping packets in bursts was done in order to detect potential re-ordering. Host Domain Name System (DNS) look-up was disabled to ensure that we only sampled the network RTT to the web-sites themselves.

As seen in the results in appendix C, while the RTT is very different for each host, the variance of the RTT per host is quite small except for a few hosts. The Google sites have sub-millisecond response time on average, but when inspecting the IP addresses, we see that this is because `ping` is in fact getting an echo reply from local web caches and not from the Google web servers. In addition, some hosts did not respond to the echo request at all, most likely due to firewall settings. Packet loss and re-ordering is not included in the table since for all of the 100 hosts, either all 1000 ping packets arrived in-order or none did.

If we filter out the hosts that either did not reply or are biased because they are web caches, we see that for a total of 60 hosts, loss and re-ordering did not occur and RTT variance was relatively small. We have, however, sampled too little data to say anything conclusive. The time of day often determines the amount of traffic to a web server, running tests for a couple of minutes at an arbitrarily chosen time is not very realistic. In addition to this, the Open Shortest Path First (OSPF) advertisement interval is 30 minutes meaning that the tests would not necessarily have run long enough to be affected by route changes [83]. Even though our tests are too limited to say anything conclusively, they still provide us with an indication of how common these error conditions are.

3.3 Test environment

We decided that the best approach in order to conduct our fairness experiment was to create a networking environment using real hardware and recent versions of Linux. We argue that using actual hardware

[2] `http://www.alexa.com/topsites`

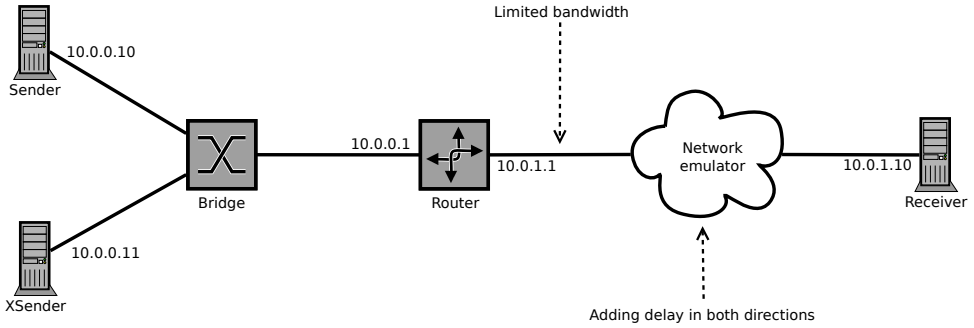


Figure 3.2: Testbed network topology

rather than virtualised hardware makes the environment easier to predict and understand, because it eliminates any potential pitfalls surrounding side-effects introduced by the virtualisation software. An additional argument was that the hardware that was available to us simply did not have enough computing power to virtualise multiple instances of Linux. Also, using real hardware allows for evaluation of problems caused by hardware offloading on the network interface controllers (NICs).

3.3.1 Network topology

Figure 3.2 depicts the topology of the network testbed we use for our experiment. The hardware and system specifications for the hosts are listed in appendix B. Traffic is sent from two senders to a single receiver. The host labelled Sender generates thin-stream traffic, while the host labelled XSender generates cross-traffic. We decided to use two hosts rather than one because of limitations with the number of threads and processor count. This is described in section 3.3.5.

Similar to a home network or a data centre, the end-hosts used the router as their preconfigured gateway rather than using Address Resolution Protocol (ARP) to resolve the end-host addresses. The senders and the receiver were put on two separate /24-subnets — opposed to having all hosts on the same Local Area Network (LAN) — although this had no practical implications for the tests.

In order to evaluate the fairness of the thin-stream modifications, we need some sort of competition over a shared limited resource, i.e. a link with reduced capacity. This was achieved by making the router a bottleneck by reducing the bandwidth of the outgoing link. We also used a network emulator to add realistic networking delays. The router configuration and network emulator are explained in section 3.3.3 and 3.3.4 respectively.

3.3.2 Traffic control

As mentioned in section 3.2.3, routers use queues to temporarily buffer packets. In Linux, each network interface has an outbound **queuing discipline (qdisc)** associated to it, which is an object consisting of a

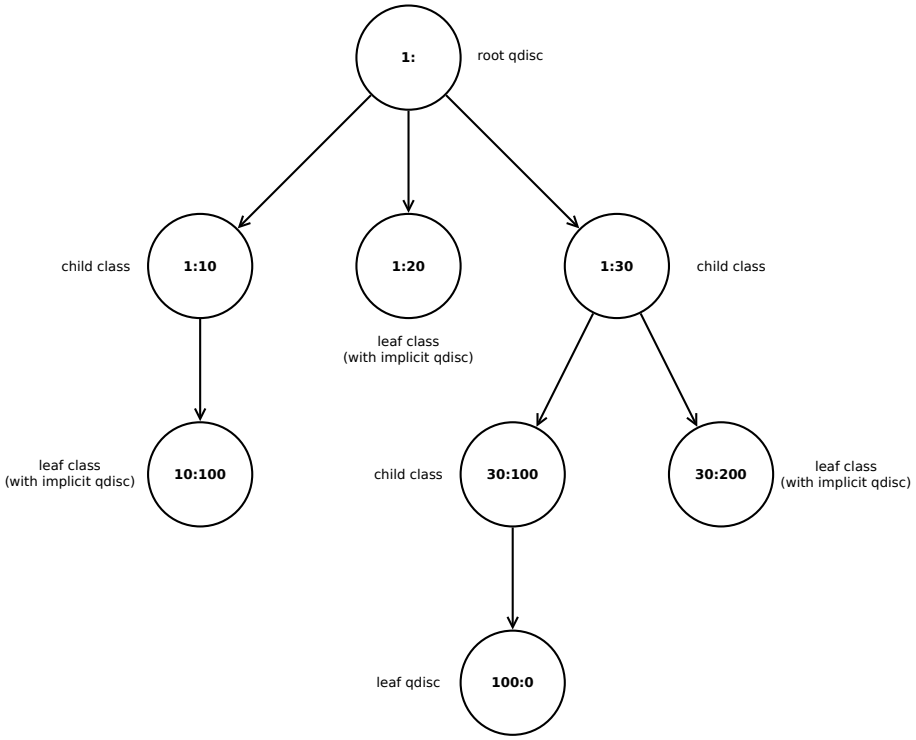


Figure 3.3: Classes and qdiscs. Note the *major*:*minor* numbering system.

buffer (queue) and a scheduler. These qdiscs can be configured and managed through the traffic control (`tc`) tool, which is offered as part of the `iproute2` utility collection [46] [125, 126]. Qdiscs can be either classless or classful. A classful qdisc can contain *classes*, and every class has a handle to which you can attach either multiple children classes or a single child qdisc. Classes only exist inside qdiscs, and a class without a child class or qdisc is a leaf class. Classless qdiscs do not have classes.

Classes are very flexible, and makes it possible to create advanced traffic control scenarios. Figure 3.3 shows how you can make a complex traffic control tree. Leaf classes have an implicit FIFO qdisc. Note that each qdisc and class are given a unique identifier on the form *major*:*minor*. The major number can follow any arbitrary numbering scheme, but all objects sharing the same parent must use the same major number. If the minor number is 0, this tells the system that the object is a qdisc — any other number identifies the object as a class.

Only leaf qdiscs actually buffer the packets, and the system uses the root qdisc as its interface. The two key interfaces are enqueueing packets in the buffer, and dequeueing packets in order to release it from the buffer and push it to the network device. Figure 3.4 demonstrates this. Packets are coming in over the wire, and the system calls `enqueue()` on the root qdisc. The root qdisc delegates this to its children qdiscs. Dequeuing packets are done in the same way: the system calls `dequeue()` on the root qdisc, and the root is responsible for calling `dequeue()` on one of its children qdiscs.

Classes can have filters associated to them as well, making it possible to create powerful packet classification schemes which can be used for shaping, policing and marking. Qdiscs, classes and filters

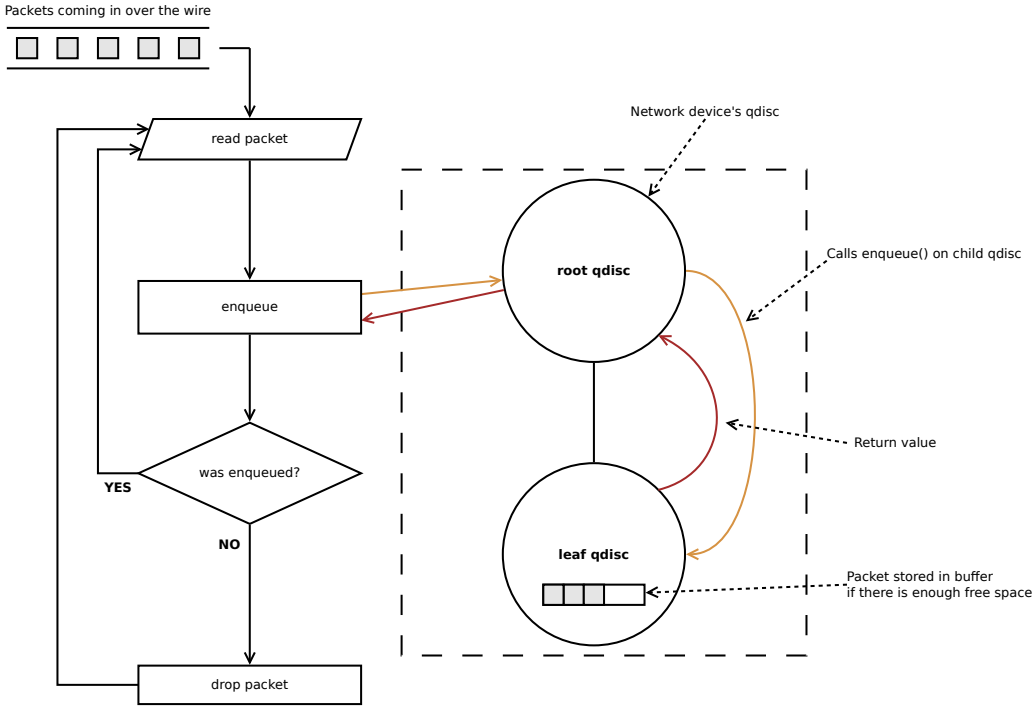


Figure 3.4: Packet enqueueing in a hierarchical qdisc configuration

makes it possible to implement advanced QoS regimes using `tc`. `tc` comes with a set of various qdiscs such as stochastic fair queueing (`sfq`), controlled delay (`codel`) and random early detection (`red`). [30, 31, 43, 46, 47] [125, 126]

3.3.3 Router configuration

The router is an integral part of our testbed, depicted in figure 3.2. It acts as the bottleneck point, responsible for creating competition between the traffic streams. The capacity of the outgoing link, shown as the link between the router and the network emulator, is reduced using a rate control implementation on the router. We also configure a queue used for buffering packets when the router starts getting saturated, as discussed in 3.2.3. In this section, we will explain how both are accomplished using `tc` and qdiscs.

Rate control

Limiting the capacity of a network link can be done by capping the bandwidth, either in hardware or in software. The ideal is of course doing this in hardware, since it reduces the number of pitfalls we could potentially fall in to [86]. On Linux, configuring the NIC is done through the `ethtool` utility program. However, very few Ethernet NICs support speeds other than the three most common Ethernet variants: 10BASE-T, 100BASE-TX and 1000BASE-T which are 10 Mbps, 100 Mbps and 1 Gbps respectively [30]. In order to run tests using a bandwidth cap lower than 10 Mbps, we decided to use a software based rate control instead which allowed us to limit the bandwidth at an arbitrary rate.

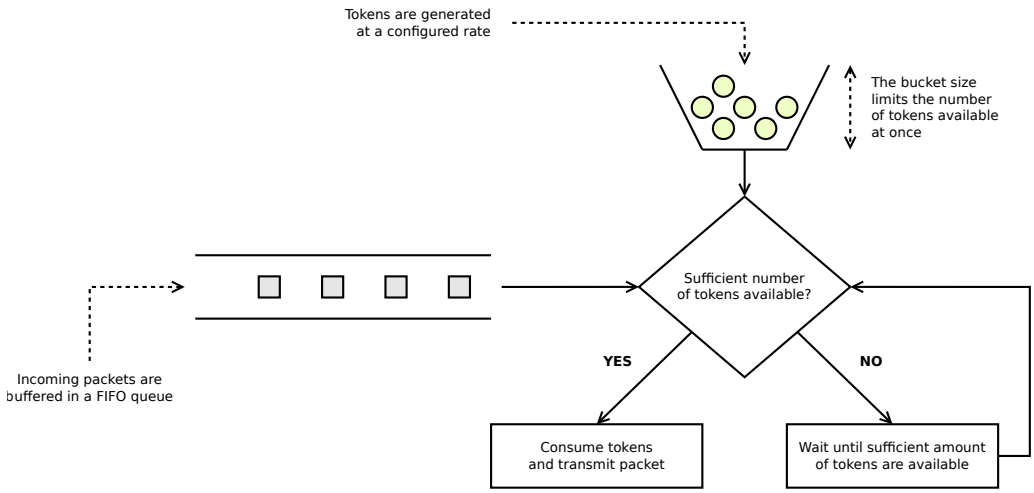
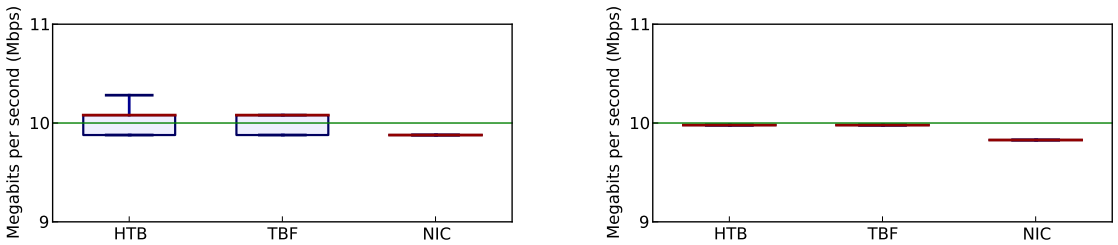


Figure 3.5: Token bucket algorithm for rate control

One way of doing rate control in software, is to use a token bucket algorithm. Figure 3.5 shows how this works. Tokens are generated at a steady rate, R, and put into a bucket. For simplicity, assume that one token represents one byte. Whenever a packet of size S is dequeued and pushed down to the network, S tokens are also consumed. If there are less than S tokens in the bucket, the packet has to wait until there are a sufficient amount of tokens available. The bucket is never filled beyond its maximum size, which means that the depth of the bucket, B, determines how bursty network traffic is. E.g., if B is $4 \times$ MTU then a maximum of four MTU-sized packets can be sent in a burst.

`tc` has two token bucket implementations, namely hierarchical token bucket (`htb`) [127] and token bucket filter (`tbf`) [128]. A rate control not based on a token bucket algorithm has also been implemented in `netem` [46]. However, since both `tbf` and `netem` seems to change from classful to classless and back again from kernel version to kernel version [129, 130], as well as the `netem` rate control being quite new [46], we used `htb` for our bandwidth limitation.

We evaluated `htb` against both `tbf` and a hardware rate control in order to verify that the rate control worked as intended, testing how they performed at 10 Mbps. Figure 3.6 shows the result of a test of the rate control implementations. We limited the rate of the outgoing link to 10 Mbps using `htb`, `tbf` and the NIC respectively. The queue in front of the rate control was set to 1514 bytes (slightly more than an



(a) Kernel clock HZ=250

(b) Kernel clock HZ=1000

Figure 3.6: Rate limitation accuracy at different clock granularities

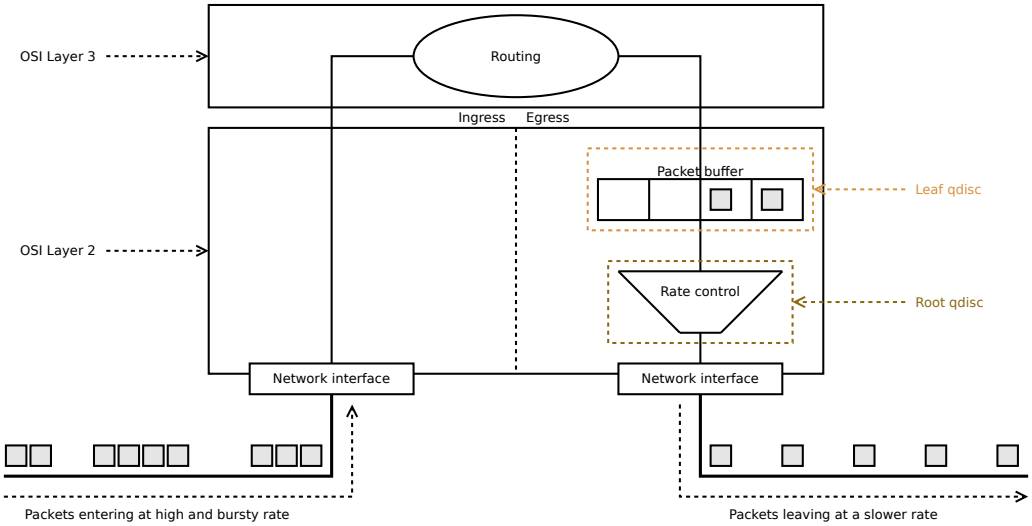


Figure 3.7: Packet flow through a Linux router with rate control

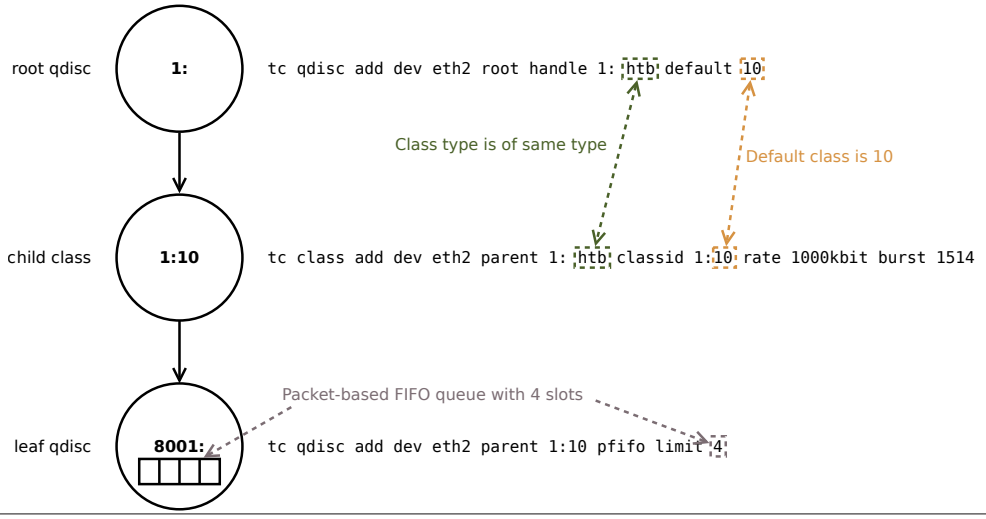
Ethernet MTU) in all three cases. We used a command line utility called `iperf` to create an UDP stream that attempted to transmit 20 Mbps from the sender to the receiver through the router for 1 minute. The green line shows exactly where the 10 Mbps mark is.

As seen in 3.6a, the two token bucket implementations are not accurate, while all samples for the the NIC rate control test are below the line. Investigations pointed at the kernel clock tick rate being the problem as many Linux distributions uses 250 hertz for the system timer [46] [86]. After recompiling the kernel on the router and setting the frequency to 1000 hertz, which is the finest granularity the clock can have on the x86 architecture [46] [131], we saw that the bandwidth rate became more accurate (see figure 3.6b).

Packet buffering

The simplest form of buffering is a simple FIFO queue, where packets are taken out of the buffer in the same order they are put in. Figure 3.7 depicts a packet's flow through a Linux router, and outlines the order of rate control and buffering. Since `htb` is a classful qdisc, we can add FIFO queue as a leaf qdisc of `htb`. Linux offers three FIFO qdiscs: packet-based FIFO (`pfifo`), byte-based FIFO (`bfifo`) and a variant called `pfifo_fast`. `pfifo` is a queue with a fixed number of slots, each slot can hold one packet. `bfifo`, on the other hand, has a fixed number of bytes it can hold, and the actual number of packets that can be held in the queue depends on the packet sizes.

The last variant, `pfifo_fast`, is the default qdisc which is used by leaf classes and interfaces without a configured qdisc [46] [125]. It is similar to `pfifo`, but it also provides some prioritisation. Internally it uses three FIFO queues, which are called *bands*, and uses the DiffServ Code Point (DSCP) field in the IP header to determine which band to place the incoming packet in. However, `pfifo_fast` does not offer any configuration possibilities to the user. Because of this, we have chosen to not include `pfifo_fast`




```
1 tc qdisc add dev eth2 root handle 1: htb default 10
2 tc class add dev eth2 parent 1: classid 1:10 htb rate 1000kbit burst 1514
3 tc qdisc add dev eth2 parent 1:10 pfifo limit 4
```

Listing 3.1: `tc` command example

in our evaluations because we are unable to configure how it prioritises traffic and the queue length it uses.

We focus mainly on `pfifo` in our experiment, since studies show that FIFO queues is still the most employed form of buffering [2,43,44]. From a fairness perspective, `pfifo` is the most relevant since all packets are treated equally. In some tests, we used other qdiscs such as `bfifo`, `red`, `codel` and `sfq`. These tests are presented in chapter 5.

Listing 3.1 demonstrates how we can limit the rate of a link to 1 Mbps and use a `pfifo` queue with four packet slots for buffering using `tc`. The first command specifies that the interface's root qdisc should be a `htb`, and that it should use class 10 as the default class for packets. The second command adds a `htb` class as a child of the root qdisc, specifying the rate tokens are generated (`rate`) and the bucket size (`burst`). We specify a burst of just slightly more than an Ethernet MTU. The third and final argument adds a `pfifo` queue with room for four packets. Note that the handle of the leaf qdisc is chosen by the system because we did not specify a handle ourselves.

Although criticised [43, 44] [84], packet buffers are usually configured to reflect the BDP. The theory is that the queue length should allow a TCP stream to fill the pipe [43, 48]. For our tests, we therefore used the BDP to calculate the queue lengths — using both the BDP calculated from one-way delay time (OWD) as well as the RTT — unless otherwise is specified.

3.3.4 Network emulator

As discussed in section 3.2.5, a network emulator was needed in order to emulate realistic network conditions. To accomplish this, we used the Linux network emulator (`netem`). `netem` is implemented as a classful qdisc, which allows it to be used in advanced networking scenarios. It has support for delaying traffic, re-ordering packets and dropping packets based on a random distribution. [46] [121]

In order to evaluate the effects of the thin stream modifications, we configured our testbed to have a fixed network RTT for all streams originating at the sender, with a symmetrical delay each direction. Ideally, all TCP traffic should have different network RTTs in order to avoid `cwnd` growing and decreasing synchronous. However, we argue that an equal RTT is necessary for a *fairness* test when we consider the fairness definition used by TCP implementations, such as NewReno (see section 2.7.1. Because of this, traffic from the sender have the same RTT, while traffic from the cross-traffic sender have different RTTs. We accomplish this by using the priority scheduler (`prio`) qdisc in combination with `netem`, using the `filter` utility to assign traffic to a band (class) based on IP address and port and having different `netem`s attached to the leaf classes.

Albeit inconclusive, our findings in section 3.2.5 suggests that re-ordering, jitter and loss not caused by congestion are not that common in the modern Internet, and that these effects usually occur at the end-points [45].

3.3.5 Traffic generation

We use a program called `streamzero` in order to create both thin streams and greedy streams. The program mimics a thin-stream sender by trying to send small amounts of data and sleeping for an interval before it repeats the process. In other words, `streamzero` transmits packets of a given size at given intervals. It can also generate greedy streams depending on the bottleneck capacity, by writing larger chunks of data to the socket descriptor and sleeping for shorter intervals. As seen in table 2.1, packet sizes and IAT varies slightly. In order to simulate this, it is also possible to specify that `streamzero` should vary its ITT and packet sizes choosing a random value from a normal distribution, where the user specifies the mean and standard deviation. Using `streamzero` means that we were able to create deterministic traffic patterns.

As `streamzero` uses several threads to create multiple streams, we use two sender hosts in order to avoid complications due to hardware limitations; both senders are dual core machines supporting hyperthreading, which means that they are able to run only four threads simultaneously. The specification for each host is listed in appendix B.

In order to make sure all connections are established before transmitting data and saturating the network, each `streamzero` thread waits on a barrier synchronisation primitive after successfully calling the `connect()` socket function. When the threads are woken up, they can be sure that all connections have been established successfully. In addition to this, we also use a ramp-up interval between each stream starting up in order to avoid a large number of streams attempting to transmit simultaneously.

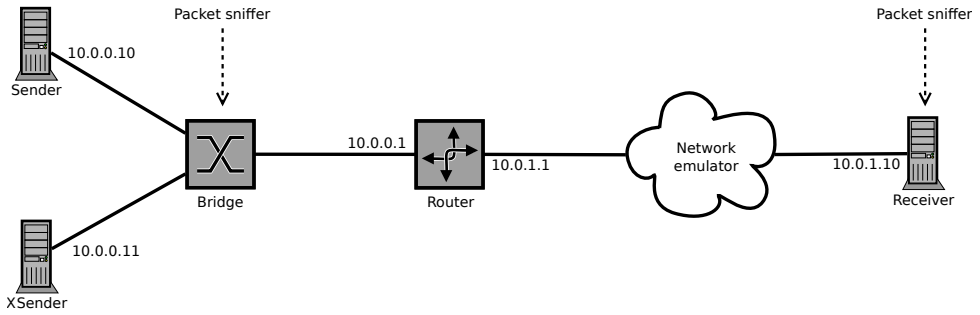


Figure 3.8: Traffic capturing in our testbed

This ramp-up interval can either be a fixed interval, e.g. 100 milliseconds between each stream start-up, or can be selected from a normal distribution with a fixed mean value and a specified standard deviation. This allows for some randomness in order to avoid inter-stream synchronisation.

Since the thin stream modifications work with both TCP CUBIC and TCP NewReno, we run tests using both. These two are the most common TCP variants, and are available by default in a vanilla Linux kernel.

3.4 Analysis tools

An important part of our evaluation is how we measure and analyse data. In this section, we review the tools and analysis software we use for our analysis of the experiment data and how they related to the fairness metrics we discussed in section 3.1.

3.4.1 `tcpdump`

Our main data source was the use of packet sniffers on the sender-side and receiver-side of the testbed. We used a packet sniffer utility program called `tcpdump` [132]. This program captures network traffic sent and received on an interface. The captured packets can then be stored to file, using the `pcap` format. These packet trace files can be read by a program, by using the `libpcap` library.

Packets stored in `pcap`-format include a timestamp of when it was captured. However, keeping system clocks synchronised for longer periods of time is a difficult task due to clock drift [24]. To circumvent this, we use a bridge between the two senders and the router, as shown in figure 3.8. This allows traffic from both the senders to be aggregated at the bridge, and the timestamps are synchronised.

3.4.2 `tcp-throughput` and `tput`

Bandwidth share allocation is the traditional fairness metric for TCP congestion avoidance mechanisms [10–13] [49]. Because of this, we decided to look at the bandwidth share each connection is allocated by

examining their throughput and goodput aggregated over time intervals.

Note the difference between throughput and goodput. Throughput is the number of bytes that a stream is able to transmit on the wire. Goodput is the number of bytes useful to the application.

For measuring goodput, we used the `tcp-throughput` tool developed by the authors of the previous thin stream evaluations [4]. This program reads a sender-side `pcap` trace file, and divides a stream's duration into *time slices*. The program keeps track of the payload size of the sent packets. For every ACK it encounters in the trace file, it calculates the corresponding time slice based on the capture timestamp, and stores the number of bytes the ACK acknowledged in that time slice.

In order to measure throughput, we use a self-developed utility program called `tput` [133]. The idea is similar to `tcp-throughput` — it stores the throughput aggregated over time intervals — but it is much simpler because it only looks at the packets being sent. Instead of reading the sender-side trace file, `tput` reads the receiver-side trace file. For each stream, the stream's duration is divided into time slices as above. Then the program iterates over all packets belonging to that stream. For each packet, it calculates the time slice it belongs to based on the timestamp, and stores the packet size (including the size of all headers) in that time slice.

3.4.3 `analyseTCP`

Another `pcap` analysing tool we used, was a program called `analyseTCP`. This program was originally created by the authors of the previous thin stream evaluations [4, 15], but we have improved it and made some modifications to it in order to make it more reliable, as well as adding new functionality. The specific contributions are listed in the end of this section.

The main benefit of `analyseTCP` over any other `pcap`-analyser is that it focuses on *latency* in many ways. No other `pcap`-analysing tool we know of provide such statistics. `analyseTCP` accepts two packet traces; a sender-side and a receiver-side trace. This allows `analyseTCP` to determine actual packet loss rather than estimating it by examining retransmissions, and it also allows the program to examine the OWD.

The program starts by reading the sender-side trace. For each TCP segment it encounters in the trace file, a byte range object is created and stored in memory, mapped to a specific stream. A byte range object is metadata about the segment payload such as the timestamp when it was first transmitted, how many times it detected retransmitted and the relative timestamp when it was ACKed. Relative timestamps are calculated by subtracting the timestamp of a stream's first packet from every other packet timestamps.

The reason we register byte ranges, rather than segments, is because TCP is allowed to do repacketisation. When a retransmission is encountered for a byte range, the sent count is increased and the timestamp of the retransmission is also stored. If only a part of a byte range is retransmitted, the byte range is split into two smaller byte ranges. Linux uses a technique called segment collapsing (repacketisation) on

retransmissions[3], a setting that is enabled by default [101]. This technique is similar to Nagle's algorithm explained in section 2.5.1. When a retransmission occur, Linux will concatenate the payload of small segments in larger, MSS-sized segments.

After the sender-side trace is processed, the receiver-side is read by the program. TCP segments in the receiver-side trace is matched to the byte range objects in memory by using the timestamp found in the TCP header option. The relative received times are also stored.

Latency

Ultimately, the goal of the thin-stream modifications is to reduce the latency. In order to measure the effectiveness of the modifications, we examine application-layer delay time (ALD). ALD is the time it takes from the first time a byte range is sent until it is ACKed. In other words, we define ALD as ACK-time: the time it takes from the first time a byte range is registered in the sender-side trace, to the ACK covering it is found in the sender-side trace. This means that packet loss, retransmission delays and queueing delays will affect the ALD measurement.

Since we register the relative receive time from the receiver-side trace, we are also able to calculate the one-way delay variance (OWDVAR). Only packets that are matched in both traces (using the TCP timestamp) have their OWDVAR calculated, so unlike ALD, we only include the last time a packet was (re)transmitted and arrived successfully.

However, system clocks on two separate hosts are unsynchronised and prone to drifting[4], we must compensate for clock drift when calculating the OWDVAR. Based on observations made in the previous thin stream studies [4], we assume that clock drift is linear. The pseudo-code in listing 3.2 shows how this is done.

[3]It is also necessary for evaluating RDB, explained in section 2.6.6, which bundles segments together. This is, however, outside the scope of our thesis.

[4]Clock drift depends a lot upon the hardware and OS technology of the computer. Newer computer have much less drift than old computers.

```
diffs = <relative received timestamp - relative sent timestamp for all byte ranges>

start_diff = <infinity>
for diff in <first 200 of diffs>:
    if diff < start_diff:
        start_diff = diff

end_diff = <infinity>
for diff in <last 200 of diffs>:
    if diff < end_diff:
        end_diff = diff

drift_factor = end_diff - start_diff

for stream in <all streams>:
    first_ts = <received timestamp of first network packet in stream>
    for byte_range_object in <byte ranges belonging to stream>:
        ts = <received timestamp of byte_range_object>
        duration = ts - first_ts
        adjusted_ts = ts - duration * drift_factor
```

Listing 3.2: Pseudo-code for linear clock drift compensation

This solution is not ideal: it assumes that there are two packet within the first and last 200 packets respectively that each have the minimum delay — the network OWD — and uses this to determine the clock drift. These two packets have their OWDVAR calculated to 0. If the network is congested when the stream starts up, the calculations become skewed. Packets might even get a negative OWDVAR calculated if the network becomes less congested after the initial 200 packets. It is, however, easy to detect such anomalies by using programmatic asserts.

As we mentioned in section 3.3.5 however, the tool we use for generating traffic does not start transmitting data until all connections are established due to the threads waiting on a barrier. In practice, this means that the packet among the first 200 packets with the lowest difference in relative received and sent times is likely to be the SYN packet. Since there is no other traffic until all connections are established and there are no sources for delay, the SYN packet's sent and received difference reflects the true network OWD. In other words, because of the way we establish connections, we can with a fair amount of certainty say that the calculated OWDVAR and the clock drift compensation are correct.

Since we use a static network RTT and do not include lost packets when calculating OWDVAR, the OWDVAR is in fact the **queueing delay** of a packet. This is because the only source of delay is either at the router or in transmission buffers. However, since we capture packets on a separate bridge, we measure *after* the packet has left the transmission buffer. We also have symmetrical delay in both directions, which means that we can calculate the absolute OWD of a packet by adding the lowest calculated RTT divided by two. The lowest RTT is calculated by simply taking the smallest time between the last time a byte range was registered as sent to the time it was ACKed, and since this is done on one side only (and in other words uses the same system clock), we know this value is accurate.

Loss

As previous studies have identified retransmissions as the greatest contributing factor to transport delay [4–6], an important metric is loss. Because `analyseTCP` uses two traces, it is able to produce accurate loss statistics. Packet loss is calculated by identifying packets that are found in the sender-side trace but not in the receiver-side trace. As with latency calculation, segments are identified with the TCP timestamp option. Byte-based loss is calculated by comparing the number of bytes received with the number of bytes attempted sent. In addition, `analyseTCP` also estimates loss based on retransmissions.

However, we are also able to distinguish between new and old data being lost, by looking at how retransmissions are lost. With this, we can provide accurate numbers for how many n-th retransmissions a stream has, as well as providing statistics about the dupACKs. These are useful for evaluating LT and MFR respectively.

Our contributions to `analyseTCP`

In order to make `analyseTCP` more reliable, we compared its output to the output of other packet trace analysing programs, such as `tcptrace` [134], Wireshark and `tshark` [135] [136] and `captcp` [137]. Even though these only provide limited latency and loss statistics, compared to `analyseTCP`, we were still able to verify our calculations by comparing statistics the other programs did offer, such as RTT statistics, estimated loss, retransmissions, number of special packets (SYN, FIN, pure ACKs). In situations where the results from `analyseTCP` differed from the other three, we investigated and corrected the calculations. In situations where the output from the three differed between themselves, we investigated why and made `analyseTCP` produce the results we deemed the most correct.

A major improvement was to make `analyseTCP` use relative sequence numbers instead of absolute, and making sure it handled sequence number wrapping correctly. The tests for checking whether a given sequence number is a wrapped number or a previous number out of order were copied from the Linux kernel implementation for managing sequence numbers in TCP.

Another contribution was to extend the already existing OWD calculation with queueing delay calculation and more accurate clock drift compensation. Queueing delay was calculated by filtering out segments that were lost.

The old version of `analyseTCP` offered crude loss rate statistics: packet-based loss and estimated loss by looking at retransmissions. We added calculation of loss over time, aggregating loss into time slices similar to how we calculate throughput. In addition, we also distinguish between lost retransmissions and new data being lost, thus improved on the old functionality that simply counted 1st through n-th retransmissions of a byte range. We also use the timestamp found in TCP header options (which is on by default in Linux) to match ranges in the sender and receiver trace more accurately.

Throughput and goodput calculation was also implemented in `analyseTCP`. Throughput is calculated in the exact same way it is in `tput`. Goodput is calculated similarly as in `tcp-throughput`, where

ACKs encountered in the sender-side trace are inspected.

3.4.4 aqmprobe

As part of our process of verifying the output of `analyseTCP`, we implemented a simple kernel module, dubbed `aqmprobe` [138], in order to inspect the router queue in real-time. By using a kernel probe (Kprobe) [139], it attaches itself to the `pfifo` entry point, the `pfifo_enqueue()` function, and records some metadata about the packet such as stream, packet size and whether or not it was dropped. A user-space program can extract these records by reading periodically from a character device the kernel module registers.

The module starts by registering a kernel return probe (Kretprobe) on `pfifo_enqueue()`. Under the hood, the module replaces the first instruction at the memory location of `pfifo_enqueue()` with an x86 breakpoint instruction. When the CPU hits this instruction, a trap occurs. The registers are saved, and the control is passed to the Kretprobe which executes the handler function, passing the saved registers as arguments. In other words, as a `struct sk_buff`-pointer and a `struct Qdisc`-pointer are passed as arguments when `pfifo_enqueue()` is invoked, we intercept this call and are able to record some data about the packet being attempted enqueued in the qdisc.

`pfifo_enqueue()` returns a success value indicating whether or not the packet was enqueued in the qdisc or not. By replacing the return address with an address of a second handler function, Kretprobe is able to intercept when functions return as well. In other words, we have a second handler function that is executed when `pfifo_enqueue()` returns, allowing us to intercept the return value and record whether or not the packet was dropped or successfully enqueued.

Passing data to the user-space application is done through a multiple-producer, single-consumer message queue. Constraining the module to only allow a single user-space program to read from the character device, means that we were able to use atomic compare-and-swap operations on the message queue and can avoid using spin-locks completely. This is important, as packet enqueueing happens quite frequently and our probe must not disrupt the service as this would skew our results.

3.4.5 count3way

In some tests, where congestion was extreme, we experienced that Linux would often drop connections that were not fully established by a complete three-way handshake (see 2.4.2) and retry establishing the connection after some time[5]. In all of the `pcap` analysis programs we used, these reconnected connections were counted as new, separate connections. In order to verify our connection statistics, we made an utility program called `count3way` that identified streams by host addresses and port numbers alone, and counted the number of successful and unsuccessful TCP three-way handshakes performed by a specific stream.

[5]The default SYN retry timeout in Linux is 20 seconds [101].

3.5 Summary

In this chapter we outlined the design of our fairness experiment. In order to evaluate the fairness of the thin-stream modifications, we selected three metrics to test our hypotheses stated in section 1.4:

- Bandwidth allocation
- Latency
- Loss

We also discussed the design of our test environment. A Linux networking testbed was created, using a bottlenecked router in order to create contention between streams over a shared resource. For generating traffic, we use a program called `streamzero`. This program establishes TCP connections, and allow us to generate network traffic with similar patterns seen in table 2.1.

We also discussed some considerations surrounding choosing realistic network conditions for our tests. By conducting a naïve `ping` test, we observed that sporadic loss, jitter and packet re-ordering is not that common in the Internet. Packet loss is often caused by network congestion, so the most realistic approach to generate loss events is to have competition over a shared, limited resource. For this reason, we implemented a bandwidth limitation in our testbed using a router with rate control.

In the next chapter, we will present the results of from an iterative *try — improve — retry* process of finding the right parameters and conditions for our tests.

Chapter 4

Verifying the testbed

While studies of the impact of increased aggressiveness in TCP in general is not something new, the behavioural patterns of thin streams are not well understood. Evaluation of thin streams is mostly uncharted territory, and there are few guidelines to follow and no best practices to refer to. Therefore, experimentally assessing a reasonable trade-off between improved thin stream performance and the effects of increased thin-stream aggressiveness on the system, depends greatly on how our test environment is implemented and how well our analysis tools work. Because of this, it is of utmost importance that we eliminate concerns and limitations surrounding our testbed and test scenarios.

4.1 A naïve approach

In order to test hypotheses 1 and 2 in particular, we design a simple fairness test based on the fairness test done in the previous thin stream evaluations [4, 13]. We let N thin streams, using the LT and/or MFR retransmission modifications, compete with N greedy streams. As our control test, we let N thin streams using unmodified retransmission mechanisms compete with N greedy streams. By comparing the results we should be able to both determine how effective the thin stream mechanisms are in increasing thin stream performance, as well as the impact these mechanisms have on greedy stream traffic.

We configured our testbed accordingly: N thin stream senders competing with N greedy stream senders. The bottleneck link speed was limited at 1 Mbps, and the network RTT was set to 100 milliseconds using `netem`. These values were chosen because it yields a BDP of 12,500 bytes, which is defined as the limit for an LFN in RFC 1072 [73]. We configured `pfifo` with a queue length of 9 packets to be used on the router. This corresponds to the BDP rounded up to the nearest whole packet ($12,500/1500 \approx 8.33$). The thin stream senders were configured to transmit packets with a 120 byte payload every 100 millisecond (172 bytes IP packets including IP and TCP headers as well as SACK header and TCP timestamp). An interval similar to the RTT meant that the thin stream senders would alternate between sending one packet within an RTT and none at all. As we had no prior estimates of how congested the network would be or how high the loss rates would become in our test scenario, we ran each test for an hour. We deemed

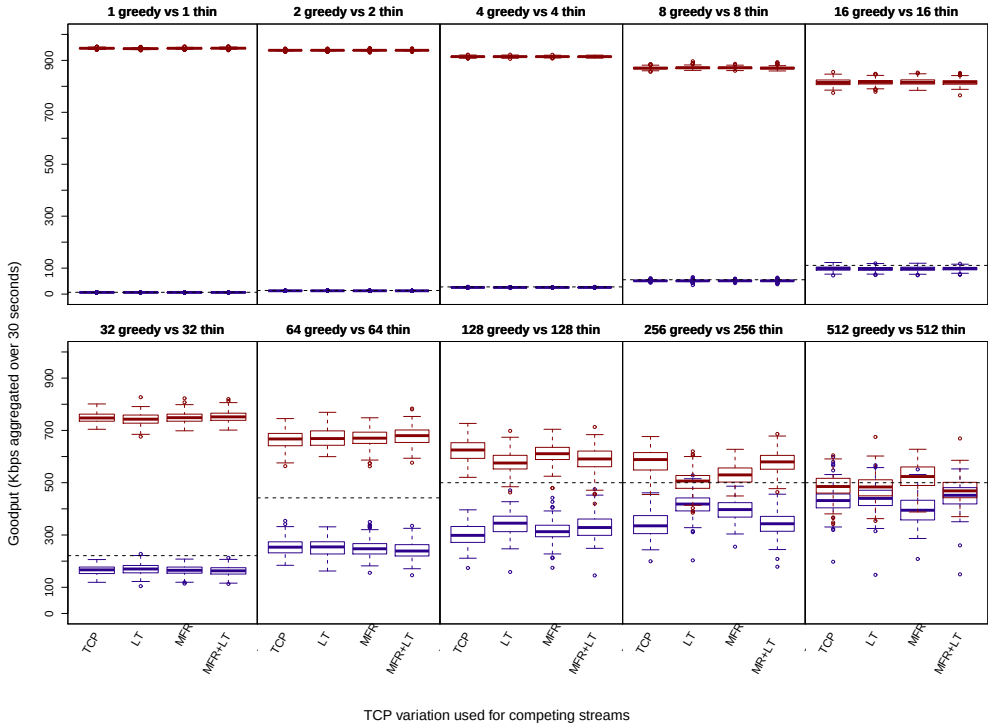


Figure 4.1: Aggregated goodput of N thin streams (blue) competing with N greedy streams (red)

this to be more than long enough to produce both enough loss events for the less congested scenarios and for the connections in the more congested scenarios to reach a stable rate.

Figure 4.1 shows the goodput of N thin streams (blue) and N greedy streams (red) aggregated over 30 second intervals. For each N, we compared LT, MFR and LT combined with MFR. We can see that as N grows, the goodput of both types of streams converges. LT, both alone and in combination with MFR, appears to have slightly more impact than MFR alone when congestion increases. We speculate that this is because the number of successive retransmissions becomes more significant as the congestion grows.

The stapled line represents the expected goodput the thin streams should have, based on how much they attempt to transmit. We observe that as the competition increases, the thin streams become less and less able to maintain their expected goodput. Note that for $N = 256$ and $N = 512$, the expected goodput is limited by sharing the capacity with a number of competing streams. We also see that the goodput varies somewhat and appears less predictable for the two most congested scenarios ($N = 256$ and $N = 512$). We suspect that this is caused by the high loss rate experienced under congestion.

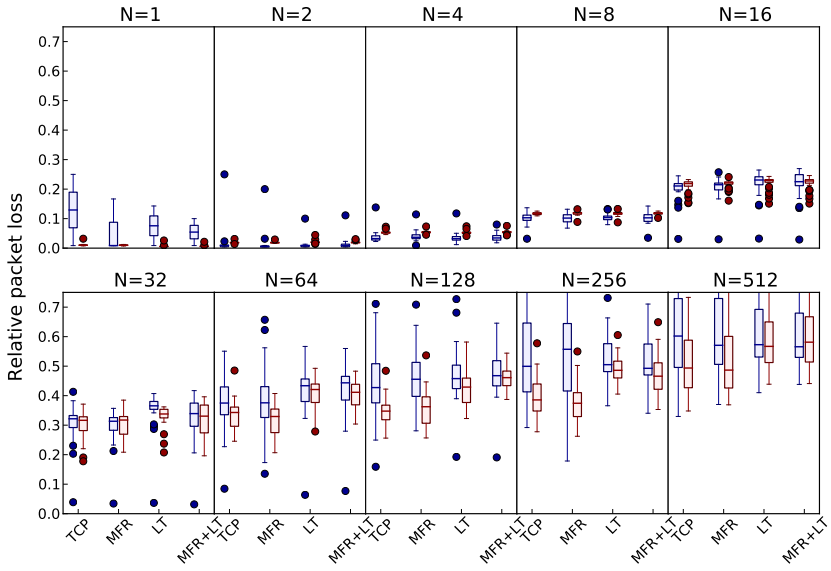


Figure 4.2: Relative loss for N thin streams (blue) competing against N greedy streams (red)

N	Dropped	No data	Never est.	Total
...	...	...	...	...
64	43	43	43	44
128	66	76	76	84
256	188	192	176	209
512	403	411	397	445

Table 4.1: Number of problematic thin stream connections

4.1.1 Too congested?

Figure 4.2 shows the relative packet loss for N thin streams (blue) competing with N greedy streams (red). As we see in the figure, thin streams have a higher overall loss rate than the greedy streams. We suspect that this might be why thin streams are unable to maintain their expected throughput. Note that as congestion builds up, being more aggressive results in a higher loss rate for other streams, as seen in $N = 64$, $N = 128$, $N = 256$ and $N = 512$.

As we see in the figure, the loss rates are very high. The median loss rate is approximately 55% for thin streams for $N = 512$. By running shorter, ten minute tests, we also saw that the data spread increases. This is a red flag, as this indicates that the connections are taking too long in order to reach a stable rate.

When looking at the number of unestablished connections, connections that last shorter than half of the test duration (dropped connections), and connections where no data packets were received due to the connections being in a state where they continuously back off, we see that for $N > 64$ more than half of the connections are problematic. This is because the level of congestion is actually quite severe; as the router is the bottleneck, with a maximum queue length of 9 packets and 512×2 streams simultaneously competing over a slot in the queue, there will always be at least $1024 - 9 = 1015$ streams that will not

have a packet in flight at any given time. We also see that for the most congested scenario, $N = 512$, most connections do not even complete the TCP three-way handshake.

4.2 A thought-through approach

As our previous approach turned out to have several problems, such as extreme congestion and packet loss, which rendered most of our findings doubtful to say the least, we went back to the drawing board. Instead of determining the queue size from an arbitrarily chosen BDP, we chose a typical household Asymmetric Digital Subscriber Line (ADSL) downlink speed for our bottleneck link rate. Get, a common Norwegian ISP, offers 5 Mbps as their most affordable speed [140].

Since the mechanisms we evaluate assume that thin streams have few PIF, our network RTT can not be too low as this would mean that we would have to transmit packets more often. Therefore we configured a network RTT of 150 milliseconds, 75 milliseconds delay in each direction[1]. This gave us a BDP of 93,750 bytes, or approximately 63 MTU-sized IP packets.

As we observed in the previous section, even though the results may look relevant when looking at a single measure, it is important to include multiple metrics in order to conclusively determine whether the results are valid or not. By looking at a multitude of measures, we were able to find scenarios were results were reproducible without having to run tests for a long time. Table 4.2 shows that the results from running the same scenario twice, two hours and five minutes respectively, scales according to the run time.

4.3 Thin stream discrimination

Looking at table 4.2, we can see that the loss rates for thin streams are somewhat more spread than the greedy stream loss rates. When we combine this observation with the observation we made in section 4.1, that thin streams are unable to maintain their expected goodput, we started suspecting that there might be some effect that was somehow inhibiting thin streams. In this section, we describe our process in identifying the culprit of the apparent unfair treatment of thin streams.

4.3.1 Comparing throughput and goodput

As we discussed in section 2.7, the traditional fairness principle of TCP is based on the assumption that each TCP stream tries to consume as much of the available bandwidth as possible. Since thin streams do not behave this way, they can not achieve a fair share of the bandwidth by this definition of fairness.

Duration_Test_2hours

Duration_Test_2hours	120 min 0 sec
Network parameters	
BW	5 Mbps
RTT	150 ms
BDP	63 pkts
Queue	pfifo 63 pkts
Cross traffic, greedy	
Conns	10
Ramp-up	100 ± 0 ms
Evaluated traffic, thin	
Conns	40
Ramp-up	100 ± 50 ms
ITT	100 ± 0 ms
Pkt size	120 ± 10 B
No RetrClps	Off
No Nagle	On
No DelAck	On
ER	On
ER+	Off
TLP	Off
LT	Off
MFR	Off

Duration_Test_2hours

Duration_Test_2hours : 1 / 2	**Cross traffic, 10 greedy streams**									
	min	Q1	med	Q2	max	1%	5 %	90%	95%	99%
Pkts sent	270.7K	275.0K	277.6K	279.6K	284.7K	270.7K	271.0K	280.5K	282.6K	284.3K
Pkts lost (%)	2.5	2.6	2.6	2.6	2.7	2.5	2.5	2.7	2.7	2.7
Bytes lost (%)	2.5	2.6	2.6	2.6	2.7	2.5	2.5	2.7	2.7	2.7
Calc loss (%)	2.5	2.6	2.6	2.6	2.7	2.5	2.5	2.7	2.7	2.7
Pkts retr	7.1K	7.2K	7.2K	7.3K	7.4K	7.1K	7.1K	7.3K	7.3K	7.4K
Successive retr	3.0	3.0	4.0	4.0	4.0	3.0	3.0	4.0	4.0	4.0
DupACKs	51.6K	53.0K	53.3K	53.9K	54.2K	51.7K	52.2K	54.0K	54.1K	54.2K
Connections dropped										None

Duration_Test_2hours : 2 / 2	**Evaluated traffic, 40 thin streams** *No Nagle;No DelAck;ER*									
	min	Q1	med	Q2	max	1%	5 %	90%	95%	99%
Pkts sent	28.7K	28.7K	28.7K	28.7K	28.8K	28.7K	28.7K	28.8K	28.8K	28.8K
Pkts lost (%)	5.5	5.6	5.6	5.6	5.7	5.5	5.5	5.7	5.7	5.7
Bytes lost (%)	5.0	5.1	5.2	5.2	5.3	5.0	5.1	5.2	5.3	5.3
Calc loss (%)	5.5	5.6	5.6	5.6	5.7	5.5	5.5	5.7	5.7	5.7
Pkts retr	1.6K	1.6K	1.6K	1.6K	1.6K	1.6K	1.6K	1.6K	1.6K	1.6K
Successive retr	2.0	2.0	2.0	3.0	5.0	2.0	2.0	5.0	5.0	5.0
DupACKs	1.0	1.0	1.0	2.0	3.0	1.0	1.0	2.0	2.2	3.0
Pkt size	82.0	236.0	254.0	357.0	1.4K	209.0	220.0	379.0	722.0	858.0
ITT (ms)	12.0	230.0	239.0	248.0	31.5K	206.0	217.0	258.0	436.0	447.0
Trans delay (ms)	75.0	132.0	144.0	156.0	219.0	103.0	115.0	167.0	172.0	182.0
ACK delay (ms)	150.0	230.0	239.0	248.0	31.3K	206.0	216.0	258.0	667.0	696.0
Connections dropped										None

Duration_Test_5mins

Duration_Test_5mins	5 min 0 sec
Network parameters	
BW	5 Mbps
RTT	150 ms
BDP	63 pkts
Queue	pfifo 63 pkts
Cross traffic, greedy	
Conns	10
Ramp-up	100 ± 0 ms
Evaluated traffic, thin	
Conns	30
Ramp-up	100 ± 0 ms
ITT	100 ± 0 ms
Pkt size	120 ± 0 B
No RetrClps	Off
No Nagle	On
No DelAck	On
ER	On
ER+	Off
TLP	Off
LT	Off
MFR	Off

Duration_Test_5mins

Duration_Test_5mins : 1 / 2	**Cross traffic, 10 greedy streams**									
	min	Q1	med	Q2	max	1%	5 %	90%	95%	99%
Pkts sent	10.0K	11.8K	11.9K	12.1K	13.0K	10.2K	10.8K	13.0K	13.0K	13.0K
Pkts lost (%)	2.2	2.4	2.6	2.7	3.1	2.2	2.2	2.8	3.0	3.1
Bytes lost (%)	2.2	2.4	2.6	2.7	3.1	2.2	2.2	2.9	3.0	3.1
Calc loss (%)	2.2	2.4	2.6	2.7	3.1	2.2	2.2	2.9	3.0	3.1
Pkts retr	280.0	294.0	308.5	312.5	336.0	280.5	282.7	325.2	330.6	334.9
Successive retr	2.0	2.0	2.0	2.0	4.0	2.0	2.0	3.1	3.5	3.9
DupACKs	2.0K	2.2K	2.2K	2.4K	2.5K	2.0K	2.0K	2.4K	2.4K	2.5K
Connections dropped										None

Duration_Test_5mins : 2 / 2	**Evaluated traffic, 30 thin streams** *No Nagle;No DelAck;ER*									
	min	Q1	med	Q2	max	1%	5 %	90%	95%	99%
Pkts sent	1.1K	1.2K	1.2K	1.2K	1.2K	1.1K	1.2K	1.2K	1.2K	1.2K
Pkts lost (%)	2.3	2.9	3.2	3.4	4.4	2.3	2.4	3.6	3.7	4.2
Bytes lost (%)	2.2	2.8	3.0	3.2	4.3	2.3	2.5	3.6	3.6	4.1
Calc loss (%)	2.4	3.0	3.2	3.4	4.4	2.4	2.5	3.7	3.8	4.2
Pkts retr	28.0	35.0	37.5	40.0	50.0	28.3	29.5	42.2	44.5	48.6
Successive retr	1.0	1.0	1.0	1.8	2.0	1.0	1.0	2.0	2.0	2.0
DupACKs	1.0	1.0	1.0	2.0	3.0	1.0	1.0	3.0	3.0	3.0
Pkt size	120.0	240.0	360.0	360.0	1.4K	120.0	240.0	360.0	360.0	840.0
ITT (ms)	2.0	247.0	258.0	267.0	2.9K	150.0	217.0	275.0	280.0	463.0
Trans delay (ms)	75.0	151.0	165.0	176.0	213.0	76.0	112.0	185.0	190.0	197.0
ACK delay (ms)	150.0	247.0	259.0	267.0	2.3K	150.0	217.0	275.0	280.0	720.0
Connections dropped										None

Table 4.2: Duration test, 2 hours and 5 minutes. Evaluated traffic: 40 thin streams; Cross-traffic: 10 greedy streams.

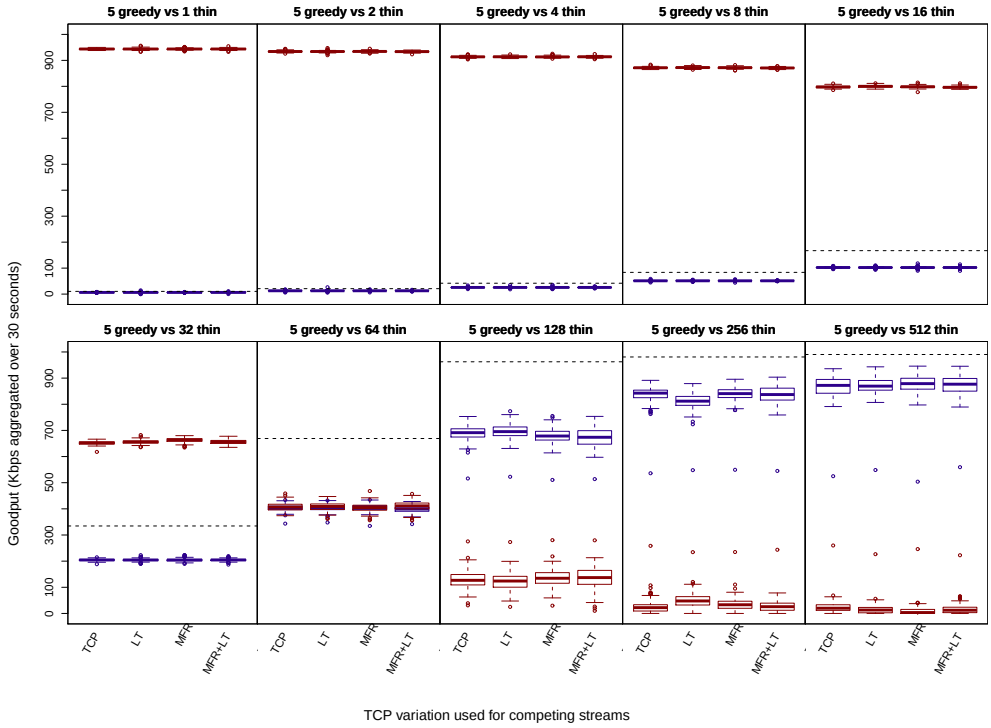


Figure 4.3: Goodput of N thin streams (blue) competing against 5 greedy streams (red)

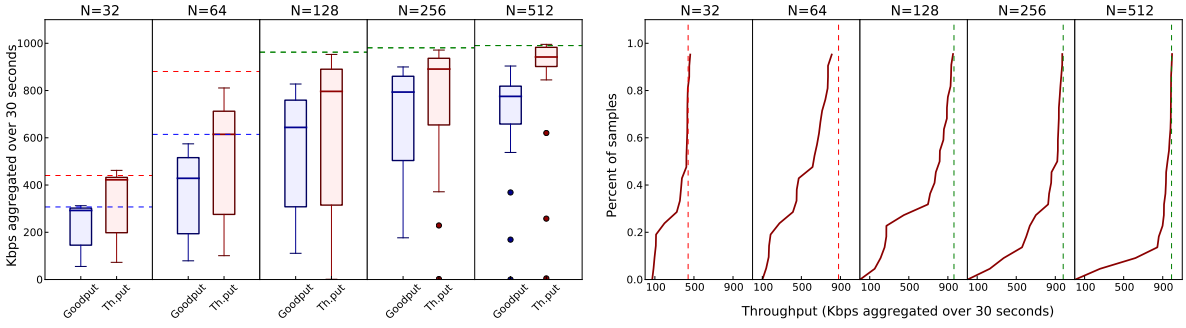
Therefore, our first suspicion was that the greedy streams were consuming an unfair share of the capacity. In order to determine if the greedy streams were to blame, we conducted a simple experiment.

Figure 4.3 shows the result of 5 greedy streams (red) competing with N thin streams. Just like in figure 4.1, we see that the thin streams struggle more and more as N increases. However, as the number of greedy streams remain constant, we suspected that the increased competition among thin streams themselves could be a possible culprit and that the greedy streams had little to do with this. In order to test this, we compare both the throughput and goodput of the thin streams. Figure 4.4 shows a subset of the results from redoing the 5 greedy versus N thin stream test[2].

The stapled blue and red lines depicts the expected goodput and throughput respectively, whereas the stapled green line is where the goodput catches up to the throughput because they are limited by the capacity. We see that neither the goodput nor the throughput reach their expected rates. As the throughput measure also includes retransmissions, this is an indication that thin streams alone are not the sole culprit and that we need to investigate this further.

[1] 150 milliseconds is roughly the RTT from Oslo to San Francisco.

[2] We had to redo the test because the data files from the one-hour tests were lost.



(a) Thin stream goodput (blue) and throughput (red)

(b) Thin stream throughput

Figure 4.4: Comparing goodput and throughput for N unmodified thin streams competing against 5 greedy streams

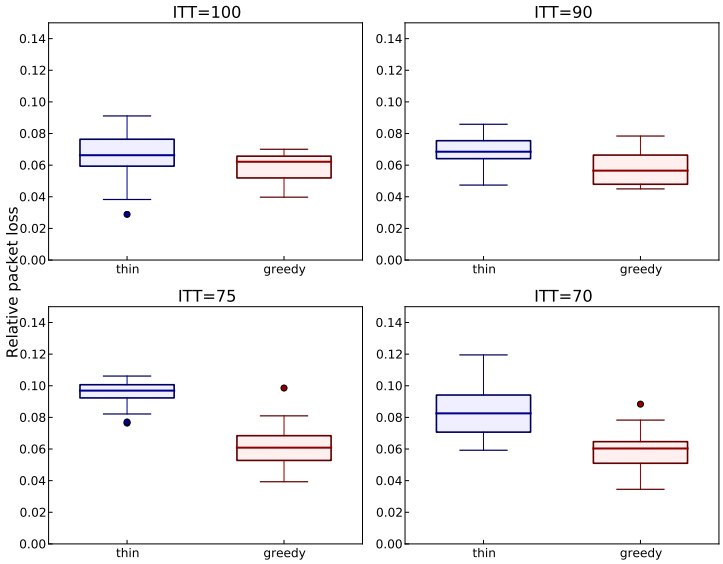


Figure 4.5: Relative packet loss for 30 thin streams and 20 greedy streams

4.3.2 Examining the loss rates

As we saw in figure 4.2 some thin streams are consistently getting higher loss rates. Arguably, a higher loss rate for thin streams, even just for some, would be unfair. Lost packets has to be retransmitted, and as we discussed in chapter 2, retransmission delays is by far the biggest contributor to transport latency.

In order to investigate this further, we decided to do a new set of tests. Figure 4.5 shows the relative packet loss rate for 30 thin streams competing against 20 greedy streams, repeated four times with different ITTs. We used different ITTs in order to rule out any unforeseen synchronisation of a badly chosen ITT and the RTT. In addition, we also chose stream ramp-up times with a slight random variation. In all four cases, some of the thin streams are consistently getting higher loss rates than the competing greedy streams.

As thin stream loss rates seemed to get worse with an increased level of congestion, as seen in figure 4.5,

we again suspected the greedy streams. It is a known effect that when multiple TCP streams using the same algorithm and with the same RTT experience loss, they back off and increase their transmission rate at the same pace [30, 31]. This is commonly known as global TCP synchronisation. We suspected that this might cause the greedy streams to alternately fill up the router queue and backing off. We moved the greedy stream senders to the cross-traffic sender machine and configured `netem` so that each greedy stream had 5 milliseconds longer RTT than the previous, starting at 100 milliseconds for the first stream. However, as the problem persisted and the results remained similar, it was clear that this was not the case.

4.3.3 Kernel buffering and repacketisation

In our packet traces, we discovered that retransmitted thin stream packets were consistently two to three times larger than the size of the original transmission. We learned that this was a result of the `tcp_retrans_collapse` system option, which is by default enabled [101]. As this, as well as Nagle's algorithm and various other settings, are a result of Linux preferring to send filled up segments rather than many small segments, we investigated how this option affected thin streams. However, as our findings varied wildly — sometimes turning it off led to higher loss rates, other times leaving it on resulted in higher loss – we decided to keep it on, as this is the default setting.

We searched for other mechanisms in the kernel that might have influenced the results, and discovered that various NIC offloading mechanisms were enabled. Even though this should be irrelevant for our testbed, since we have a bridge between the sender and the router, based on our experience with `tcp_retrans_collapse`, we turned it off in order to avoid any potential problems. Which NIC supports which offloading technique varies, but the four relevant mechanisms are: generic segmentation offload (GSO), TCP segmentation offload (TSO), generic receive offload (GRO) and large receive offload (LRO).

4.3.4 Thin stream clustering

By using the `aqmprobe` kernel module, described in section 3.4.4, we were able to analyse loss events over time. By highlighting loss after stream type, we identified a clustering of drop events that occurred on roughly every ITT for the thin streams. In other words, because we had used a too low ITT variance, thin streams clustered together. Choosing a sensible ramp-up time did not, contrary to our expectations, alleviate this effect. We reasoned that this was because the ITT is periodic and smaller than an RTT, and a ramp-up time chosen from a random distribution is not guaranteed to be spread out over the entire ITT interval.

We confirmed this clustering effect when we reduced the queue size to half its original size, and suddenly that the loss rates for roughly half of the thin streams increased. This is because as they transmit periodically, any given thin streams end up roughly in the same spot in the clustering every ITT, meaning

that some streams consistently get data through because they are enqueued properly, while other streams consistently hit a full queue (and the packet is subsequently dropped).

4.4 Summary

In this chapter we saw how seemingly meaningful results can be completely skewed by a hidden factor. We identified some culprits of thin stream discrimination, as well as a clustering effect, causing multiple thin streams to time after time experience loss because they end up in the same "bad spot" every transmission interval.

Because of time-constraints and a old computer, we have not been able to include all results and plots in this (and the next chapter). For a complete list of results, please refer the URL given in appendix A

Chapter 5

Summarising the results

As thin streams often are a product of real-time applications, they have entirely different requirements and acceptable limits than other network traffic. Where some scenarios are considered edge cases and can be excused for some applications since they happen so rarely, edge cases can be make or break for a time-dependent application.

5.1 Queue length evaluations

As part of our evaluation, we also did an evaluation of the effects of the queue length on thin streams. We configured our test scenario according to table 5.1. 30 thin streams using unmodified retransmission mechanisms competing against 30 greedy streams. We tested the impact `pfifo` and `bfifo` had on three different lengths: half a BDP, a BDP and twice the BDP.

Figure 5.1 shows the relative byte loss, number of bytes lost divided by number of bytes sent, for both thin (blue) and greedy (red) streams. 30 thin streams using unmodified retransmission mechanisms competed against 30 greedy streams. The `pfifo_short`, `pfifo_bdp` and `pfifo_long` are `pfifo` qdiscs with a queue size of 31 packets (half a BDP), 63 packets (a BDP) and 125 packets (twice the BDP respectively. Note that we had to scale up the loss rates for `pfifo_short` due to the loss rate being higher than all the others. The `bfifo_short` qdisc was configured to have a queue size of $46,875$ bytes (half a BDP), while the `bfifo_bdp` qdisc was set to $93,750$ bytes (a BDP). Although our focus was to investigate the effects of tail-drop on thin streams, we included three other queue algorithms for reference: `red`, `sfq` and `codel`.

We see that when the queue size becomes too short, as seen with `pfifo_short`, the thin streams in our scenario experience high loss rates. This is because they have similar ITT and get clustered, effectively forming a burst. Since the queue is not long enough to absorb the burst, the tail gets dropped and some streams experience loss. The effects on the thin streams in terms of latency, can be seen in figure 5.2a.

qdisc-evaluation	
Duration	10 min 0 sec
Network parameters	
BW	5 Mbps
RTT	150 ms
BDP	93750 bytes (63 pkts)
Cross traffic, greedy	
Conns	30
Ramp-up	100 ± 0 ms
Evaluated traffic, thin	
Conns	30
Ramp-up	100 ± 0 ms
ITT	100 ± 0 ms
Pkt size	120 ± 0 B
No RetrClps	Off
No Nagle	On
No DelAck	On
ER	Off
ER+	Off
TLP	Off
LT	Off
MFR	Off

Table 5.1: Test configuration used for our qdisc evaluation

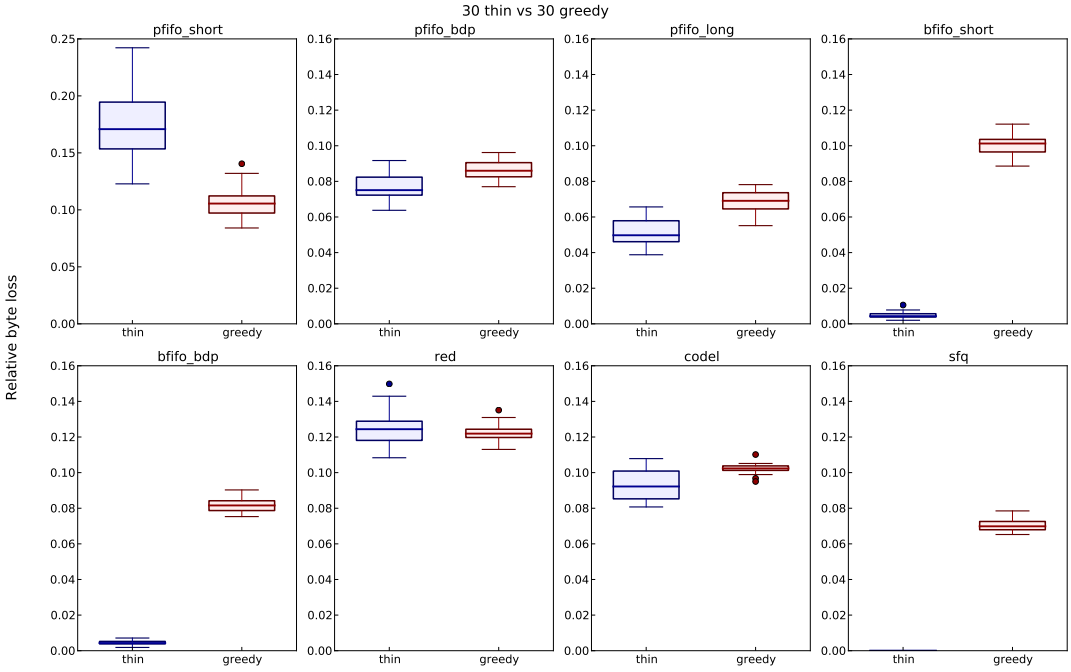
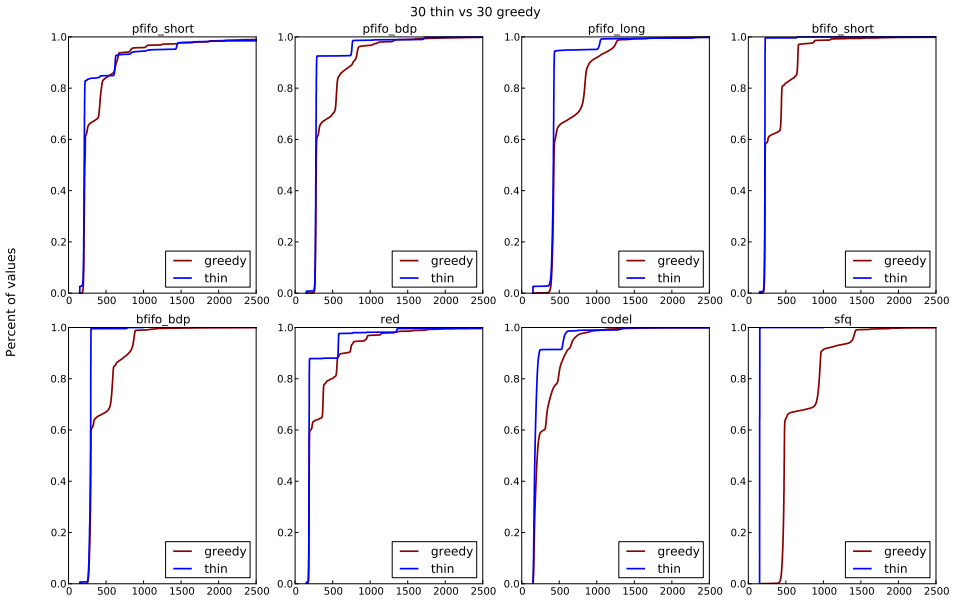
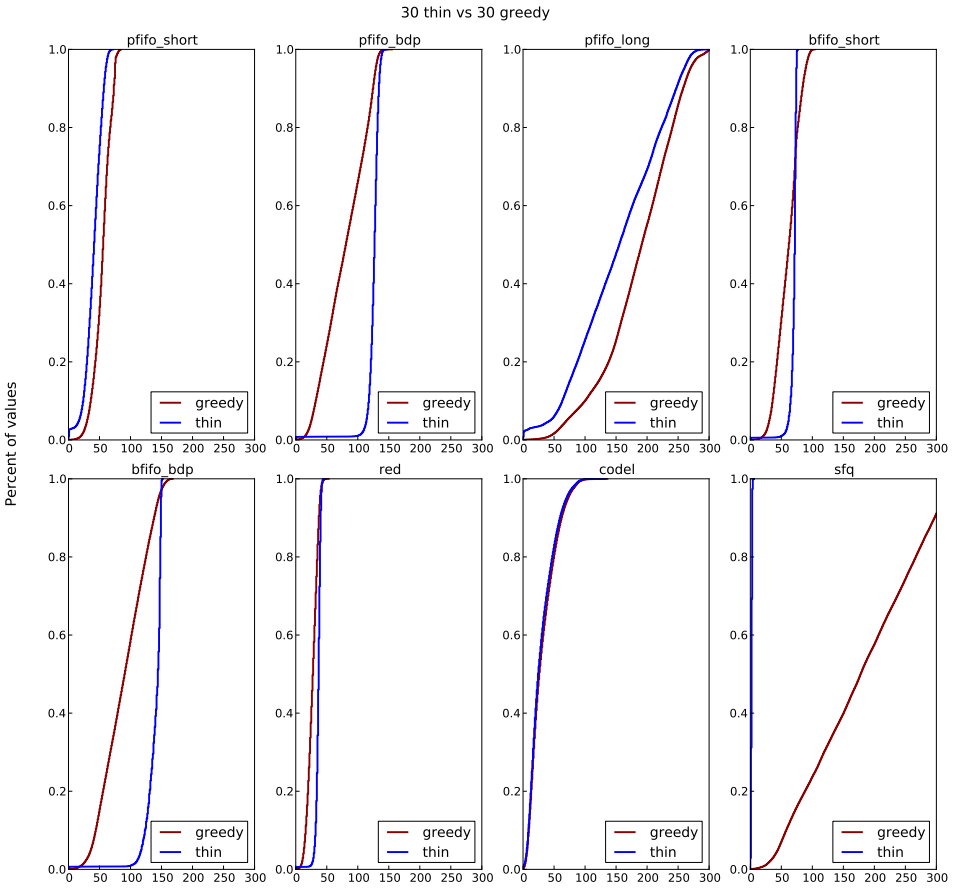


Figure 5.1: Relative byte loss using different qdiscs. 30 unmodified thin streams competing with 30 greedy streams.



(a) ACK delay



(b) Queueing delay

Figure 5.2: Impact of queue configuration on latency

With a longer queue, more of the thin stream cluster fits in the buffer, and the thin streams get better ACK latency.

However, longer queues also mean longer queueing delays. It is a rule of thumb in real-time networking to keep buffers to a minimum size in order to keep latencies as low as possible [2, 31, 43] [84, 86, 141]. As seen in figure 5.2b, for the longest `pfifo` queue, the 90th percentile is almost 4 times higher as for the shortest `pfifo` queue, which of course is a huge increase in delay. However, it reduces the overall thin stream latency because less thin streams experience loss. This is a trade-off that needs careful consideration. Many game servers and sensor networks, typical thin stream applications, operate in ticks, where they periodically send out updates. For a game server, for example, the number of clients might be very high, resulting in bursts of packets being sent out at periodic intervals. The immediate packet buffers must be large enough to not drop packets, but short enough to not delay packets unnecessary. The best configuration for such scenarios is a possible candidate for further study.

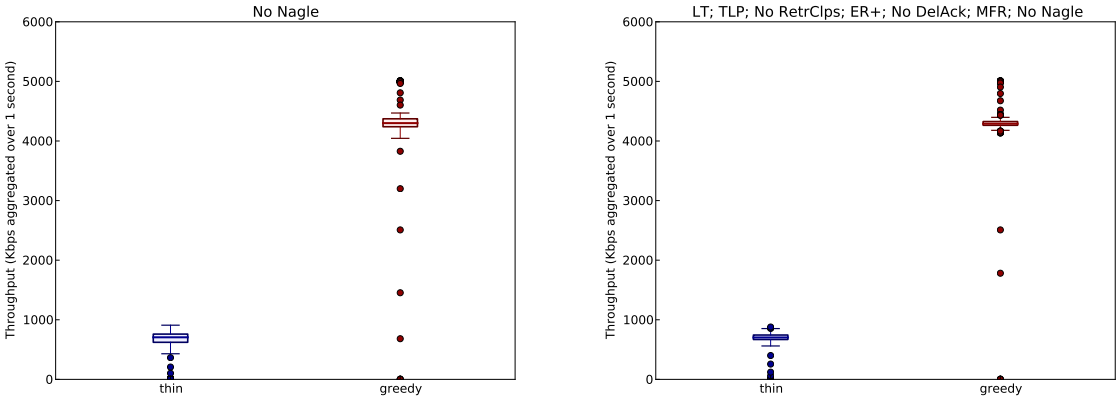
The `bfifo` queue tells a different story. Hardly any thin stream packets are dropped, even when the queue is short. This is actually because byte-based queues most often be able to fit a small packet, where larger packets are dropped. Although not as bad as the `pfifo` queues, packets enqueued in a `bfifo` are still subject to quite high queueing delays (a queueing delays as high as the RTT for the 90th percentile and above).

`sfq` is the queue that maximises the thin stream performance in our scenario. This is because of how `sfq` works: packets are hashed based on their source and destination IP address and destination port address. Based on this hash, they are put into a bucket — a virtual queue. Since thin streams and greedy streams send to their separate servers using different receiver ports, thin and greedy streams are given separate virtual queues. The benefit of this is that streams do not suffer from how other, misbehaving streams behave. However, `sfq` is not widely used in the Internet [2].

5.2 Assessing the impact on other streams

As we saw in section 4.1, it is hard to determine whether we are measuring the impact thin streams have on greedy streams, or if we are measuring the impact thin streams have on themselves. It is clear that greedy TCP streams are robust and able to to adapt other, competing network traffic. Thin streams, however, are vulnerable because they are limited by retransmissions and not by congestion control and can not recover effectively using fast recovery, see section 2.6.

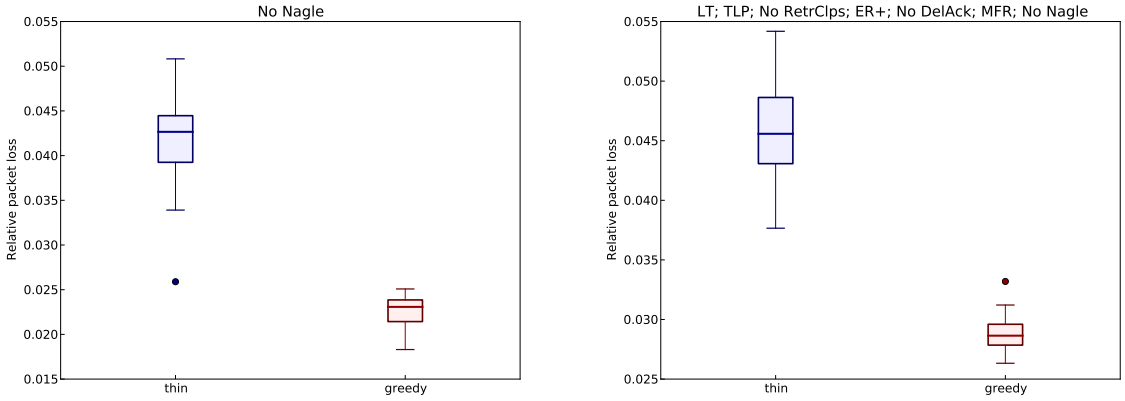
In order to determine the impact of the thin stream modifications on other streams, we did two tests, one with 60 thin streams using all modifications competed against 10 greedy streams, and the other with 60 unmodified thin streams (using none of the modifications) competed against 10 greedy streams. As seen in the following figures, 5.3, 5.4 and 5.5, only the thin streams themselves were affected; the greedy



(a) No thin stream modifications

(b) All thin stream modifications

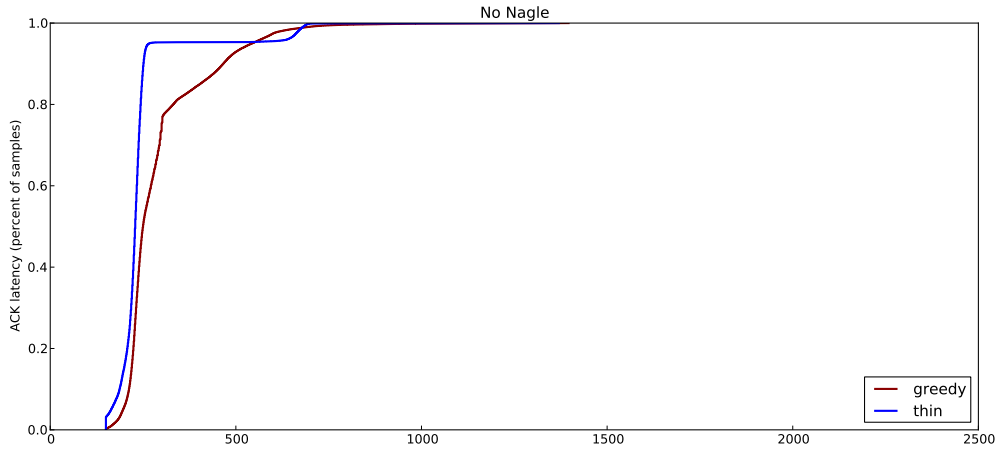
Figure 5.3: Throughput. 60 thin streams versus 10 greedy streams.



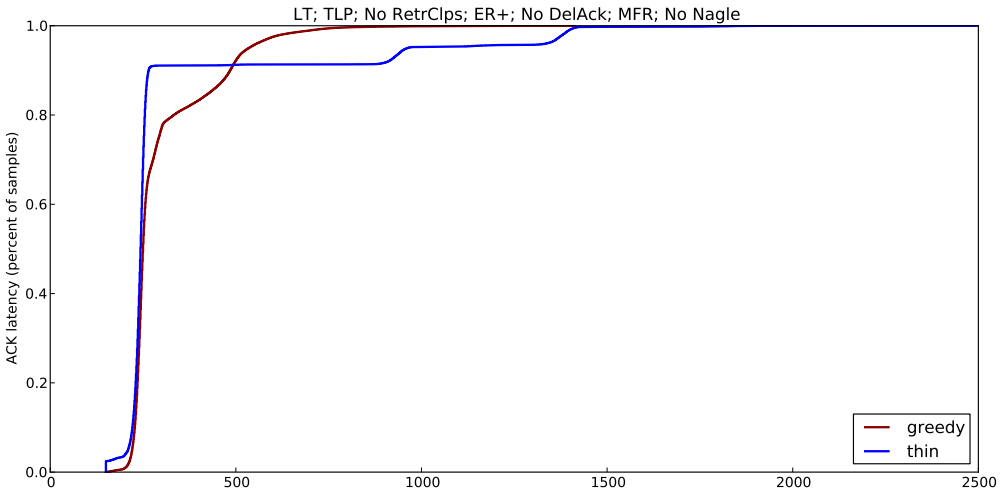
(a) No thin stream modifications

(b) All thin stream modifications

Figure 5.4: Relative packet loss. 60 thin streams versus 10 greedy streams



(a) No thin stream modifications



(b) All thin stream modifications

Figure 5.5: ACK latency. 60 thin streams versus 10 greedy streams

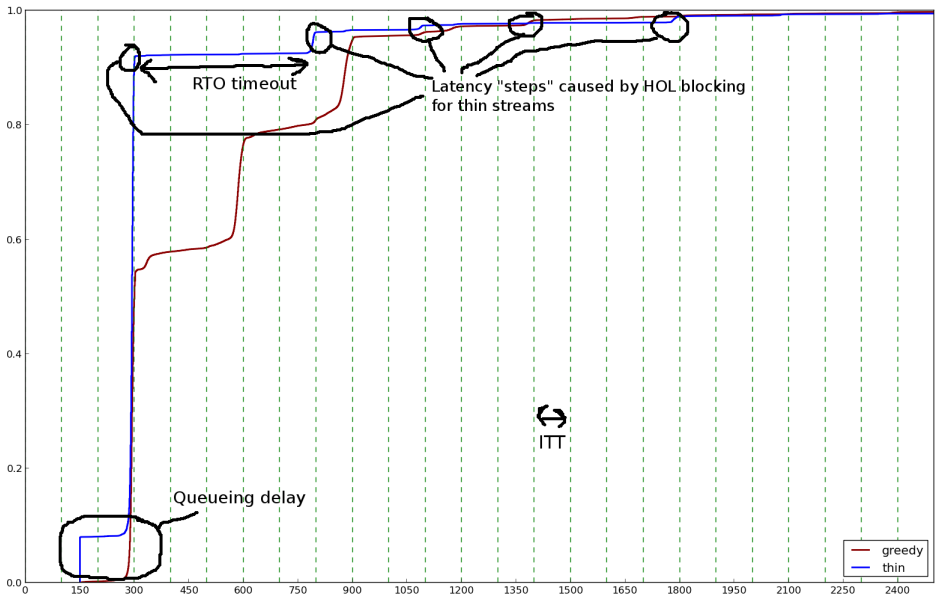


Figure 5.6: ACK latency CDF annotation

streams were able to adapt without any considerable changes in performance.

Figure 5.6 explains the various trends we observe in the ACK latency plots. We can see that the RTO has a major impact on the streams overall latency values. Also, we see that every step on the thin stream latency CDF is caused by previous unacknowledged segments blocking a new segment (HOL blocking).

Chapter 6

Conclusion

"I have not failed. I've just found 10,000 ways that won't work."

Thomas Edison

6.1 Main contributions

In this thesis, we have attempted to experimentally find the right aggressiveness for retransmissions in thin TCP streams. We have implemented a controlled test environment in the form of an emulated Linux networking testbed in order to conduct fairness experiments and evaluate the thin stream modifications to the TCP retransmission mechanisms. Our motivation for doing this has been to assess how aggressive these mechanisms can be in order to improve thin stream performance, while still remaining fair towards other traffic. As part of our evaluation, we have developed new tools and improved existing software, in order to accurately analyse thin stream performance. During this process, we have made observations that strengthens our view that an increased aggressiveness for application-limited streams can be justified.

We have identified a thin-stream clustering effect as a problem for tick-based applications (which may include games, video/audio streaming and financial applications using barriers to control transaction commits) and the detrimental effect small queues has in such scenarios, as the clusters of thin-stream packets arrive at the short queue together time after time. Some advice for providers of tick-based interactive applications to not scale for expected (average) throughput) but to scale queues to accommodate the bursts generated from the clustering effects. This finding confirms observations made in a previous thin-stream study, but which was dismissed as a simulator synchronisation effect [13].

The awareness of latency in the Internet is recently having a renaissance after many years of focus on throughput in Internet research. In this context, we have contributed to a position paper calling for a reconsideration of the traditional fairness definition and advocating increased retransmission aggressiveness for interactive thin streams in order to reduce recovery latency, published as part of the Internet

Society Workshop on Reducing Internet Latency (RITE) in September 2013. We have also made contributions to another unpublished paper on thin streams, planned to be submitted to IEEE/ACM Transactions on Networking some time in the future.

6.2 Future work

As thin stream behaviour is not well understood, further research on how thin streams behave in realistic scenarios is more than warranted. One of the previous thin stream studies attempted to classify and model different forms of thin stream [13]. We believe that a realistic model of thin streams is crucial for understanding how thin streams behave in certain circumstances.

We observed in our investigations that there are mechanisms with conflicting motives enabled by default in the Linux kernel. On one hand, there is an inherent desire to conserve packets and avoid unnecessary transmissions. On the other hand, there is an ongoing process of trying to reduce various sources of latency in the kernel, most recently addressed by the "bufferbloat" project. We believe that further investigations on mechanisms that might affect thin streams are absolutely crucial in order to increase the awareness of these contradicting mechanisms among kernel developers.

While keeping buffering to a minimum is the rule of thumb for reducing latency, we saw in our evaluations that when thin streams cluster and form bursts, having too small buffers do more harm than good. Such bursts can be common in applications that are tick-based, such as game servers or sensor networks. Finding a sweet-spot between large enough buffers to absorb these bursts and avoid loss while still keeping queuing delay is a possible candidate for future work.

Bibliography

[1] Global internet phenomena report. Technical Report 2H-2013, Sandvine Incorporated, December 2013. https://www.sandvine.com/downloads/general/global-internet-phenomena/2013/2h-2013-global-internet-phenomena-report.pdf.

[2] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the internet. *ACM Queue*, 55(1):57–65, November 2011. ISSN 1542-7730 http://queue.acm.org/detail.cfm?id=2071893.

[3] Ashvin Goel, Charles Krasic, Kang Li, and Jonathan Walpoe. Supporting low latency TCP-based media streams. In *Proceedings of the Tenth International Workshop on Quality of Service (IWQoS)*, May 2002. http://www.eecg.toronto.edu/~ashvin/publications/iwqos2002.pdf.

[4] Andreas Petlund. *Improving latency for interactive, thin-stream applications over reliable transport*. PhD thesis, Department of Informatics, University of Oslo, Oslo, Norway, October 2009. ISSN 1501-7710 https://www.duo.uio.no/bitstream/handle/10852/10152/Petlund.pdf.

[5] Espen Søgård Paaby. Evaluation of TCP retransmission delays. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, May 2006. http://www.duo.uio.no/publ/informatikk/2006/42442/Paaby.pdf.

[6] Carsten Griwodz and Pål Halvorsen. The fun of using TCP for an MMORPG. In *Proceedings of the 2006 International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '06, pages 1–7, Newport, Rhode Island, United States, 2006. Association for Computing Machinery (ACM). http://heim.ifi.uio.no/~griff/papers/funtrace.pdf.

[7] Nandita Dukkipati, Matt Mathis, Yuching Cheng, and Monia Ghobadi. Proportional rate reduction for TCP. In *Proceedings of the 11th ACM SIGCOMM Conference on Internet Measurement 2011*. Association for Computing Machinery (ACM), November 2011. http://conferences.sigcomm.org/imc/2011/docs/p155.pdf.

[8] Marco Mellia, Michela Meo, and Cladio Casetti. TCP smart framing: a segmentation algorithm to reduce TCP latency. *IEEE/ACM Transactions on Networking*, 13(2):316–329, April 2005. ISSN 1063-6692 http://research.microsoft.com/en-us/um/people/padmanab/temp/ton/fea2f08f9e3206d5001.pdf.

[9] Andreas Petlund. TCP thin-stream modifications: Reduced latency for interactive applications. *Linux Journal*, 219, July 2012. https://www.simula.no/publications/LJ-219-Jul-2012/simula_pdf_file.

[10] Van Jacobson. Congestion avoidance and control. *SIGCOMM Computer Communication Review*, 18(4):314–329, August 1988. ISSN 0146-4833 ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[11] Rajendra K. Jain, Dah-Ming W. Chiu, and William R. Hawe. A quantitative measurement of fairness and discrimination for resource allocation in shared computer system. Technical report, Digital Equipment Corporation, September 1984.
http://www1.cse.wustl.edu/~jain/papers/ftp/fairness.pdf .

[12] Bob Briscoe. Flow rate fairness: Dismantling a religion. *SIGCOMM Computer Communication Review*, 37(2):63–74, April 2007. ISSN 0146-4833
http://www.eecs.berkeley.edu/~sylvia/cs268-2013/papers/dismantling.pdf .

[13] Markus Simon Fuchs. Time-dependent thin transport layer streams: Characterization, empirical observation and protocol support. Master's thesis, Department of Computer Science, University of Kaiserslautern (TU Kaiserslautern), Kaiserslautern, Germany, January 2014.

[14] Mats Rosbach. Verification of network simulators: The good, the bad and the ugly. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, November 2012.
https://www.duo.uio.no/bitstream/handle/10852/34916/Rosbach-Master.pdf .

[15] Kristian R. Evensen. Improving TCP for time-dependent applications. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, May 2008.
http://www.duo.uio.no/publ/informatikk/2008/79691/Evensen.pdf .

[16] High-performance automated trading network architectures. Technical report, Cisco Systems Inc., 2010.
http://www.cisco.com/web/strategy/docs/finance/c11-600126_wp.pdf .

[17] Petri Arola. Algorithmic trading: Can you meet the need for speed? Technical report, Detica, 2008.
https://www.baesystemsdetica.com/uploads/resources/
d4e99b39c4c4622ce5ece31a4e2d946a1.pdf .

[18] Ultra-low-latency networking. Technical Report 888.706.4239, CDW, 2012.
http://webobjects.cdw.com/webobjects/media/pdf/Solutions/Financial/
Ultra-Low-Latency-Networking.pdf .

[19] Essential facts about about the computer and video game industry. Technical report, Entertainment Software Association, 2010.
http://www.theesa.com/facts/pdfs/ESA_Essential_Facts_2010.PDF .

[20] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11), November 2006.
http://web.cs.wpi.edu/~claypool/papers/precision-deadline/final.pdf .

[21] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in Unreal Tournament 2003. SIGCOMM' 04. Association for Computing Machinery (ACM), August 2004.
http://web.cs.wpi.edu/~claypool/papers/ut2003/ut2003.pdf .

[22] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in Warcraft III. NetGames '03. Association for Computing Machinery (ACM), May 2003.
http://web.cs.wpi.edu/~claypool/papers/war3/war3.pdf .

[23] Peter Quax, Patrick Monsieurs, Wim Lamotte, Danny De Vleeschauwer, and Natalie Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *Proceedings of ACM International Conference on Applications, Technologies, Architectures and Protocols for Computer Communication*, SIGCOMM '04 Workshops. Association for Computing Machinery (ACM), September 2004. ISBN 0-58113-042

http://conferences.sigcomm.org/sigcomm/2004/workshop_papers/net608-quax.pdf.

[24] Chris Carlmar. Improving latency for interactive, thin-stream applications by multiplexing streams over TCP. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, February 2011. http://www.duo.uio.no/publ/informatikk/2011/111955/Carlmar.pdf.

[25] One-way transmission time. Technical Report G.114, International Telecommunication Union (ITU-T), May 2003. http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-G.114-200305-I!!PDF-E&type=itemsoiceoverIP.

[26] Narrow-band visual telephone systems and terminal equipment. Technical Report H.320, International Telecommunication Union (ITU-T), March 2004. https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-H.320-200403-I!!PDF-E&type=items.

[27] Ahmed Sabbir Arif and Wolfang Stuerzlinger. Anaysis of text entry performance metrics. In *Science and Technology for Humanity (TIC-STH), 2009 IEEE Toronto International Conference*, pages 100–105, September 2009. http://www.cse.yorku.ca/~wolfgang/papers/textperfmetrics.pdf.

[28] Anuj Agarwal. High-frequency trading: Evolution and the future. Technical report, Capgemini, 2012. http://www.capgemini.com/resource-file-access/resource/pdf/High_Frequency_Trading__Evolution_and_the_Future.pdf.

[29] A. D. Wissner-Gross and C. E. Freer. Relativistic statistical arbitrage. *Physical Review E*, 82(5):056104, November 2010. http://www.alexwg.org/publications/PhysRevE_82-056104.pdf.

[30] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education International, Inc., Vrije Universiteit (VU University), Amsterdam, Netherlands, 4th edition, 2003. ISBN 0-13-038488-7.

[31] Michael Welzl. *Network Congestion Control: Managing Internet Traffic*. Communications Networking & Distributed Systems. John Wiley & Sons, Ltd., Leopold Franzens University of Innsbruck, Innsbruck, Austria, 2005. ISBN 978-0-470-02528-4.

[32] Vinton G. Cerf and Robert E. Kahn. A protocol for packet network intercommunication. In *IEEE Transactions on Communications*, volume 22. Institute of Electrical and Electronics Engineers (IEEE), Institute of Electrical and Electronics Engineers (IEEE), May 1974.

[33] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: refining TCP congestion control. In *Proceedings of ACM International Conference on Applications, Technologies, Architectures and Protocols for Computer Communication*, SIGCOMM '96, pages 281–291, Palo Alto, California, United States, 1996. Association for Computing Machinery (ACM). ISBN 0-89791-790-1 http://doi.acm.org/10.1145/248156.248181.

[34] Mohammad Alizadeh, Albert Greenberg, David Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74. Association for Computing Machinery (ACM), August 2010. ISBN 978-1-4503-0201-2 http://sedcl.stanford.edu/files/dctcp-final.pdf.

[35] Steven Low, Larry Peterson, and Limin Wang. Understanding TCP Vegas: Theory and practice. Technical Report TR-616-00, Princeton University, February 2000.
ftp://ftp.cs.princeton.edu/techreports/2000/616.pdf .

[36] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control for fast, long distance networks. Technical report, North Carolina State University.
http://www.csc.ncsu.edu/faculty/rhee/export/bitcp.pdf .

[37] Marco Mellia, Ion Stoica, and Hui Zhang. TCP model for short lived flows. *Communications Letters*, 6(2):85–87, February 2002. ISSN 1089-7798
http://iie.fing.edu.uy/ense/asign/perfredes/material_perm/shortLivedTCPFlows.pdf .

[38] Sally Floyd and Van Jacobson. On traffic phase effects in packet-switched gateways. *SIGCOMM Computer Communication Review*, 21(2):26–46, April 1991.
http://www.icir.org/floyd/papers/phase.pdf .

[39] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, Mark Stemm, and Randy H. Katz. TCP behaviour of a busy internet server: Analysis and improvements. In *Proceedings of IEEE INFOCOM*, volume 1, pages 252–262. Institute of Electrical and Electronics Engineers (IEEE), March 1998.
http://www.cs.cmu.edu/~srini/Papers/1998.Balakrishnan.infocom.pdf .

[40] IEEE standard for ethernet: Section one. Technical Report IEEE 802.3-2012, Institute of Electrical and Electronics Engineers (IEEE), December 2012.
http://standards.ieee.org/getieee802/download/802.3-2012_section1.pdf .

[41] Pasi Sarolahti and Alexey Kuznetsov. Congestion control in linux TCP. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 49–62. USENIX Association, 2002. ISBN 1-880446-01-4
https://www.usenix.org/legacy/event/usenix02/tech/freenix/full_papers/sarolahti/sarolahti_html/ .

[42] David X. Wei and Pei Cao. NS-2 TCP implementation with congestion control algorithms from Linux. In *Proceedings from the 2006 Workshop on Ns-2: The IP Network Simulator*, WNS2 '06. Association for Computing Machinery (ACM), 2006. ISBN 1-59593-508-8
http://netlab.caltech.edu/projects/ns2tcplinux/paper/wns2-final.pdf .

[43] Kathleen Nichols and Van Jacobson. Controlling queue delay. *ACM Queue*, 10(5):20–34, May 2012. ISSN 1542-7730.

[44] G. Vy-Brugier, R. S. Stanojevic, D. J. Leith, and R. N. Shorten. A critique of recently proposed buffer-sizing strategies. *SIGCOMM Computer Communication Review*, 37(1):43–48, January 2007. ISSN 0146-4833
http://www.hamilton.ie/net/ccr.pdf .

[45] Yi Wang, Guohan Lu, and Xing Li. A study of Internet packet reordering. The International Conference on Information Networking (ICOIN) 2004, pages 350–359, 2004. ISBN 3-540-23034-3
http://www.cs.princeton.edu/~yiwang/papers/icoin04.pdf .

[46] Anders Grotthing Moe. Implementing rate control in NetEm: Untying the NetEm/tc tangle. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, August 2013.
https://www.duo.uio.no/bitstream/handle/10852/37459/Moe-Master.pdf .

[47] Wu chang Feng, Kang G. Shin, Dilip D. Kandlur, and Debanjan Saha. The BLUE active queue management algorithms. In *IEEE/ACM Transactions on Networking*, volume 10. Institute of Electrical and Electronics Engineers (IEEE), August 2002. ISSN 1063-6692.

[48] Masaki Hirabaru. Impact of bottleneck queue size on TCP protocols and its measurement. *IEICE Transactions on Communications*, E89-B(1), January 2006.
http://hirabaru.org/papers/IEICE2005-final.pdf .

Internet Standards and Drafts

[49] S. Floyd. Metrics for the Evaluation of Congestion Control Mechanisms. RFC 5166 (Informational), March 2008.

[50] J. Postel. Internet Protocol. RFC 791 (INTERNET STANDARD), September 1981. Updated by RFCs 1349, 2474, 6864.

[51] JM. Valin, K. Vos, and T. Terriberry. Definition of the Opus Audio Codec. RFC 6716 (Proposed Standard), September 2012.

[52] S. Andersen, A. Duric, H. Astrom, R. Hagen, W. Kleijn, and J. Linden. Internet Low Bit Rate Codec (iLBC). RFC 3951 (Experimental), December 2004.

[53] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245 (Proposed Standard), April 2010. Updated by RFC 6336.

[54] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), April 2010.

[55] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008.

[56] J. Postel. User Datagram Protocol. RFC 768 (INTERNET STANDARD), August 1980.

[57] Audio-Video Transport Working Group, H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 1889 (Proposed Standard), January 1996. Obsoleted by RFC 3550.

[58] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (INTERNET STANDARD), July 2003. Updated by RFCs 5506, 5761, 6051, 6222, 7022, 7164.

[59] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[60] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (INTERNET STANDARD), October 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633, 6864.

[61] V. Cerf, Y. Dalal, and C. Sunshine. Specification of Internet Transmission Control Program. RFC 675, December 1974.

[62] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. Updated by RFCs 4301, 6040.

[63] N. Spring, D. Wetherall, and D. Ely. Robust Explicit Congestion Notification (ECN) Signaling with Nonces. RFC 3540 (Experimental), June 2003.

[64] J. Postel and J.K. Reynolds. Telnet Protocol Specification. RFC 854 (INTERNET STANDARD), May 1983. Updated by RFC 5198.

[65] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018 (Proposed Standard), October 1996.

[66] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323 (Proposed Standard), May 1992.

[67] V. Jacobson. Compressing TCP/IP Headers for Low-Speed Serial Links. RFC 1144 (Proposed Standard), February 1990.

[68] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire. Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry. RFC 6335 (Best Current Practice), August 2011.

[69] J.C. Mogul and S.E. Deering. Path MTU discovery. RFC 1191 (Draft Standard), November 1990.

[70] E. Blanton, M. Allman, K. Fall, and L. Wang. A Conservative Selective Acknowledgment (SACK)-based Loss Recovery Algorithm for TCP. RFC 3517 (Proposed Standard), April 2003. Obsoleted by RFC 6675.

[71] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[72] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[73] V. Jacobson and R.T. Braden. TCP extensions for long-delay paths. RFC 1072 (Historic), October 1988. Obsoleted by RFCs 1323, 2018, 6247.

[74] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, January 1984.

[75] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582 (Proposed Standard), April 2012.

[76] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Experimental), December 2003.

[77] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), April 2004. Obsoleted by RFC 6582.

[78] P. Sarolahti, M. Kojo, K. Yamamoto, and M. Hata. Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP. RFC 5682 (Proposed Standard), September 2009.

[79] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.

[80] M. Allman, K. Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig. Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP). RFC 5827 (Experimental), May 2010.

[81] M. Mathis, N. Dukkipati, and Y. Cheng. Proportional Rate Reduction for TCP. RFC 6937 (Experimental), May 2013.

[82] J. Postel. Internet Control Message Protocol. RFC 792 (INTERNET STANDARD), September 1981. Updated by RFCs 950, 4884, 6633, 6918.

[83] J. Moy. OSPF Version 2. RFC 2328 (INTERNET STANDARD), April 1998. Updated by RFCs 5709, 6549, 6845, 6860.

Internet References

[84] Jim Gettys. The criminal mastermind: bufferbloat.
http://gettys.wordpress.com/2010/12/03/introducing-the-criminal-mastermind-bufferblo
, December 2010.
Accessed July 2014.

[85] Reducing internet transport latency.
http://riteproject.eu/ .
Accessed July 2014.

[86] Best practices for benchmarking CoDel and FQ CoDel (and almost anything else!).
https://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_
benchmarking_Codel_and_FQ_Codel , March 2013.
Accessed November 2013.

[87] Actions per minute.
http://en.wikipedia.org/wiki/Actions_per_minute .
Accessed July 2014.

[88] G-series recommendations.
http://www.itu.int/rec/T-REC-G/en .
Accessed March 2014.

[89] WebRTC frequently asked questions.
http://webrtc.org/faq .
Accessed April 2014.

[90] Opus (audio codec).
http://en.wikipedia.org/wiki/Opus_(audio_codec) .
Accessed April 2014.

[91] Understanding delay in packet voice networks.
http://www.cisco.com/c/en/us/support/docs/voice/voice-quality/
5125-delay-details.html#standarfordelaylimits .
Accessed June 2014.

[92] Stuart Cheshire. Latency and the quest for interactivity.
http://www.stuartcheshire.org/papers/LatencyQuest.html , November 1996.
Accessed June 2014.

[93] Measuring video quality in videoconferencing systems.
http://www.watchpointvideo.com/pdf/Measuring%20Video%20Quality%20in%

20Videoconferencing%20Systems.pdf.
Accessed April 2014.

[94] Words per minute.
http://en.wikipedia.org/wiki/Words_per_minute.
Accessed July 2014.

[95] Richard Martin. Wall street's quest to process data at the speed of light.
http://www.informationweek.com/wall-streets-quest-to-process-data-at-the-speed-of-l
d/d-id/1054287?, April 2007.
Accessed July 2014.

[96] Viraf Reporter. The value of a millisecond: Finding the optimal speed of a trading infrastructure.
http://community.rti.com/sites/default/files/archive/V06-007_Value_of_a_
Millisecond.pdf, April 2008.
Accessed July 2014.

[97] Wikipedia. Internet protocol suite.
http://en.wikipedia.org/wiki/Internet_protocol_suite.
Accessed June 2012.

[98] Wikipedia. OSI model.
http://en.wikipedia.org/wiki/OSI_model.
Accessed June 2012.

[99] Wikipedia. Transmission Control Protocol.
http://en.wikipedia.org/wiki/Transmission_Control_Protocol.
Accessed July 2012.

[100] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. TCP fast open.
http://tools.ietf.org/html/draft-ietf-tcpm-fastopen-08, March 2014.
Accessed July 2014.

[101] IPv4 variables.
https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt.
Accessed April 2014.

[102] TCP protocol manual page.
http://man7.org/linux/man-pages/man7/tcp.7.html.
Accessed July 2014.

[103] TCP Westwood+.
http://c3lab.poliba.it/index.php/Westwood.
Accessed July 2012.

[104] M. Sridharan, K. Tan, D. Bansal, and D. Thaler. Compound TCP: A new TCP congestion control for high-speed and long distance networks.
http://tools.ietf.org/html/draft-sridharan-tcpm-ctcp-02, November 2009.
Accessed July 2014.

[105] Wikipedia. TCP congestion avoidance algorithm.
http://en.wikipedia.org/wiki/TCP_congestion_avoidance_algorithm.
Accessed July 2012.

[106] Ilpo Järvinen. RE: TCP default congestion control in linux should be newreno.
http://lists.openwall.net/netdev/2008/12/04/87.
Accessed March 2014.

[107] Kenji Kurata, Go Hasegawa, and Masayuki Murata. Fairness comparions between TCP Reno and TCP Vegas for future deployment of TCP Vegas.
http://www.isoc.org/inet2000/cdproceedings/2d/2d_2.htm.
Accessed March 2014.

[108] Neal Cardwell and Boris Bak. A TCP Vegas implementation for Linux.
http://neal.nu/uw/linux-vegas/.
Accessed March 2014.

[109] Lawrence Stewart, David Hayes, and Grenville Armitage. FreeBSD SVN commit log.
http://lists.freebsd.org/pipermail/svn-src-head/2011-February/024595.html.
Accessed March 2014.

[110] DD-WRT changelog.
http://www.dd-wrt.com/wiki/index.php/Changelog.
Accessed March 2014.

[111] Sangtae Ha and Injong Rhee. BIC and CUBIC.
http://research.csc.ncsu.edu/netsrv/?q=content/bic-and-cubic.
Accessed March 2014.

[112] Lawrence Stewart, David Hayes, and Grenville Armitage. FreeBSD SVN commit log.
http://lists.freebsd.org/pipermail/svn-src-head/2010-December/022924.html.
Accessed March 2014.

[113] CC_CUBIC manual page.
http://www.freebsd.org/cgi/man.cgi?query=cc_cubic.
Accessed March 2014.

[114] M. Mellia, M. Meo, and C. Casetti. TCP smart-framing.
http://tools.ietf.org/html/draft-mellia-tsvwg-tcp-smartframing-00, November 2001.
Accessed July 2014.

[115] Yuchung Cheng. tcp: early retransmit.
http://git.kernel.org/linus/eed530b6c67624db3f2cf477bac7c4d005d8f7ba.
Accessed June 2014.

[116] Andreas Petlund. net: TCP thin dupack.
http://git.kernel.org/linus/7e38017557bc0b87434d184f8804cadb102bb903.
Accessed June 2014.

[117] Andreas Petlund. net: TCP thin linear timeouts.
http://git.kernel.org/linus/36e31b0af58728071e8023cf8e20c5166b700717.
Accessed June 2014.

[118] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. Tail loss probe TLP: An algorithm for fast recovery of tail losses.

http://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01 , February 2013.
Accessed June 2014.

[119] Nandita Dukkipati. tcp: tail loss probe (TLP).
http://git.kernel.org/linus/6ba8a3b19e764b6a65e4030ab0999be50c291e6c.
Accessed June 2014.

[120] Jean-Yves Le Boudec. Rate adaption, congestion control and fairness: A tutorial.
http://ica1www.epfl.ch/PS_files/LEB3132.pdf , November 2012.
Accessed August 2014.

[121] netem.
http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.
Accessed June 2014.

[122] ns-3.
http://www.nsnam.org/.
Accessed August 2014.

[123] ns-2.
http://nsnam.isi.edu/nsnam/index.php/User_Information.
Accessed August 2014.

[124] Hajime Tazaki, Frédéric Urbani, and Thierry Turletti. Simulate network protocols with real stacks for better realism.
http://www.nsnam.org/wp-content/uploads/2013/01/WNS3-2013-Presentation-Tazaki.pdf , March 2013.
Accessed August 2014.

[125] Martin A. Brown. Traffic control HOWTO.
http://tldp.org/HOWTO/Traffic-Control-HOWTO/ , October 2006.
Accessed July 2014.

[126] iproute2.
http://www.linuxfoundation.org/collaborate/workgroups/networking/iproute2.
Accessed June 2014.

[127] HTB manual page.
http://linux.die.net/man/8/tc-htb.
Accessed July 2014.

[128] TBF manual page.
http://linux.die.net/man/8/tc-tbf.
Accessed July 2014.

[129] Jason Boxman. A practical guide to linux traffic control.
http://blog.edseek.com/~jasonb/articles/traffic_shaping/index.html , February 2005.
Accessed July 2014.

[130] Jarek Poplawski. sch_netem: Remove classful functionality.
http://git.kernel.org/linus/02201464119334690fe209849843881b8e9cfa9f.
Accessed July 2014.

[131] The tick rate: HZ.
http://www.makelinux.net/books/lkd2/ch10lev1sec2.
Accessed November 2013.

[132] TCPDUMP and LIBPCAP.
http://www.tcpdump.org/.
Accessed July 2014.

[133] Jonas Sæther Markussen. tput.
https://github.com/enfiskutensykkel/tput.

[134] Shawn Osterman. tcptrace.
http://www.tcptrace.org/.
Accessed August 2014.

[135] Wireshark.
http://www.wireshark.org/.
Accessed August 2014.

[136] Wikipedia. Wireshark. http://en.wikipedia.org/wiki/Wireshark.
Accessed August 2014.

[137] Hagen Paul Pfeifer. captcp.
https://github.com/hgn/captcp.
Accessed August 2014.

[138] Jonas Sæther Markussen. aqmprobe.
https://github.com/enfiskutensykkel/aqmprobe.

[139] Jim Keniston, Prasanna S. Panchamukhi, and Masami Hiramatsu. Kernel probes (Kprobes).
https://www.kernel.org/doc/Documentation/kprobes.txt.
Accessed August 2014.

[140] Get xs 5.
http://www.get.no/produkter/superbredband/xs-4.
Accessed August 2014.

[141] Stuart Cheshire. It's the latency, stupid.
http://rescomp.stanford.edu/~cheshire/rants/Latency.html, May 1996.
Accessed June 2014.

Appendix A

All test scenarios

An index of all tests performed as well as plots and datasets can be found at the following URL:
`http://folk.uio.no/jonassm/masterthesis/`

Appendix B

Testbed configuration

B.1 Specifications

Table B.1: Testbed system specifications

Host	CPU	RAM	Kernel
sender	AMD Athlon 64 X2 Dual Core 4800+	2 GB	3.12.0-rc4+ x86_64
xsender	AMD Athlon 64 X2 Dual Core 4800+	2 GB	2.6.32-5-amd64 x86_64
receiver	AMD Athlon 64 X2 Dual Core 4800+	2 GB	2.6.32-5-amd64 x86_64
bridge	Intel Pentium D 2.80 GHz	2 GB	2.6.32-5-amd64 x86_64
emulator	Intel Pentium D 2.80 GHz	2 GB	2.6.32-5-amd64 x86_64
router	AMD Athlon 64 X2 Dual Core 4800+	2 GB	3.12.0-rc3 x86_64

Table B.2: Testbed network specifications

Host	Interface	IP address	Controller	Driver	Type	Bus
sender	eth1	10.0.0.10	Marvell 88E8001	skge 1.13	1000BASE-T	PCI
xsender	eth1	10.0.0.11	D-Link DGE-528T	r8169 2.3LK-NAPI	1000BASE-T	PCI
receiver	eth1	10.0.1.10	Marvell 88E8001	skge 1.13	1000BASE-T	PCI
bridge	eth1-4	-	Intel 1521 (4 port)	igb 3.0.6-k2	1000BASE-T	PCIe x4
emulator	eth1-4	-	Intel 1521 (4 port)	igb 3.0.6-k2	1000BASE-T	PCIe x4
router	eth1	10.0.0.1	D-Link DGE-528T	r8169 2.3LK-NAPI	1000BASE-T	PCI
router	eth2	10.0.1.1	D-Link DGE-528T	r8169 2.3LK-NAPI	1000BASE-T	PCI

Since both interfaces on the router is connected to the same conventional PCI bus, the absolute maximum throughput from sender to receiver and vice versa is around 230 Mbps.

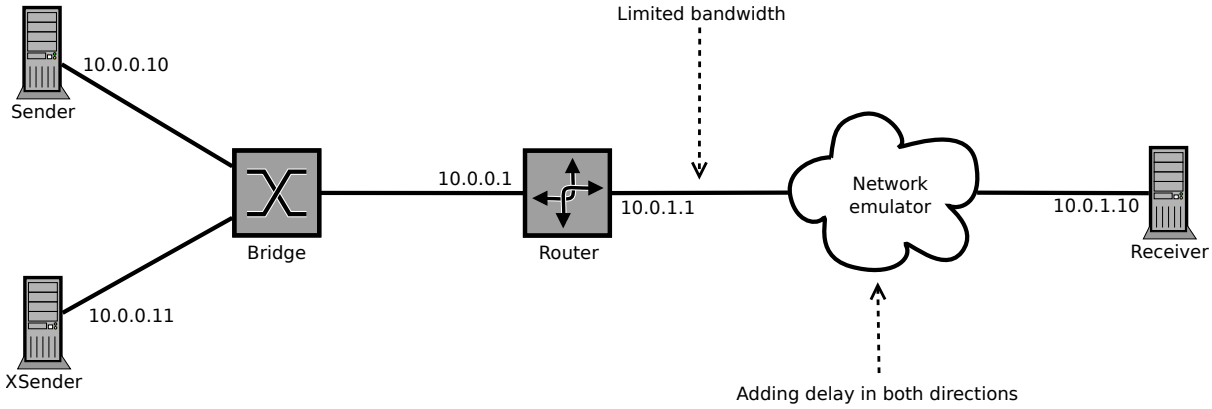


Figure B.1: Testbed network topology

B.2 Topology

We used `tc` to limit the bandwidth, `netem` as our network emulator and `brctl` for creating the network bridge. On the router, the system option `net.ipv4.ip_forward` was enabled and static routes were configured using the `ip` utility. Traffic from the sender had the same RTT, while traffic from the xsender were given different RTTs. The `txqueuelen` was set to 0 on all interfaces, and NIC offloading (TSO, GSO, TRO and LRO) was disabled.

Appendix C

Web-site RTT measurements

Table C.1: Web-site RTT measurements performed in June 2014

Host	IP address	min	avg	median	max	max-min dist
google.com	81.175.29.177	0.8	0.9	0.9	2.7	1.9
facebook.com	173.252.110.27	125.0	125.4	125.0	128.0	3.0
youtube.com	81.175.29.148	0.8	0.9	0.9	1.4	0.6
yahoo.com	98.138.253.109	138.0	144.1	145.0	148.0	10.0
baidu.com	220.181.111.86	224.0	224.0	224.0	225.0	1.0
wikipedia.org	208.80.154.224	113.0	113.0	113.0	118.0	5.0
qq.com	125.39.240.113	361.0	368.7	368.0	374.0	13.0
taobao.com	42.120.194.11	227.0	227.0	227.0	227.0	0.0
live.com	65.55.206.154	-	-	-	-	-
twitter.com	199.16.156.230	118.0	118.0	118.0	119.0	1.0
amazon.com	205.251.242.54	-	-	-	-	-
linkedin.com	216.52.242.86	185.0	185.5	185.0	187.0	2.0
google.co.in	81.175.29.154	0.8	0.9	0.9	1.0	0.2
sina.com.cn	202.108.33.60	378.0	383.0	383.0	388.0	10.0
hao123.com	180.149.131.31	225.0	225.8	226.0	226.0	1.0
weibo.com	114.134.80.162	288.0	288.0	288.0	296.0	8.0
blogspot.com	74.125.136.191	29.9	30.0	30.0	30.2	0.3
tmall.com	42.120.194.11	227.0	227.0	227.0	231.0	4.0
sohu.com	220.181.90.240	225.0	226.2	226.0	227.0	2.0
yahoo.co.jp	183.79.197.242	306.0	306.0	306.0	306.0	0.0
vk.com	87.240.131.117	21.1	21.5	21.2	28.0	6.9
yandex.ru	213.180.204.11	32.5	32.7	32.7	34.9	2.4
wordpress.com	66.155.11.243	117.0	117.0	117.0	117.0	0.0
ebay.com	66.135.216.190	-	-	-	-	-
bing.com	204.79.197.200	11.0	11.1	11.1	13.0	2.0
google.de	81.175.29.154	0.8	0.9	0.9	3.7	2.8
pinterest.com	54.225.151.74	-	-	-	-	-
360.cn	220.181.24.100	227.0	228.0	228.0	229.0	2.0
google.co.uk	81.175.29.185	0.8	0.9	0.9	3.5	2.7

Host	IP address	min	avg	median	max	max-min dist
google.fr	81.175.29.144	0.8	0.9	0.9	1.4	0.6
instagram.com	107.23.28.33	-	-	-	-	-
google.co.jp	81.175.29.166	0.8	0.9	0.9	1.0	0.2
ask.com	66.235.120.127	116.0	116.1	116.0	117.0	1.0
163.com	123.58.180.7	390.0	414.9	410.0	433.0	43.0
soso.com	220.181.124.154	224.0	224.0	224.0	227.0	3.0
msn.com	65.55.206.228	-	-	-	-	-
tumblr.com	66.6.42.20	107.0	107.2	107.0	116.0	9.0
google.com.br	81.175.29.185	0.8	0.9	0.9	1.8	1.0
mail.ru	217.69.139.199	34.6	34.7	34.7	35.7	1.1
xvideos.com	141.0.174.40	-	-	-	-	-
microsoft.com	134.170.185.46	-	-	-	-	-
google.ru	81.175.29.185	0.8	0.9	0.9	1.1	0.2
paypal.com	66.211.169.3	-	-	-	-	-
google.it	81.175.29.159	0.8	0.9	0.9	1.7	0.9
google.es	81.175.29.154	0.8	0.9	0.9	1.1	0.3
apple.com	17.178.96.59	-	-	-	-	-
imdb.com	207.171.166.22	-	-	-	-	-
adcash.com	72.52.178.205	140.0	140.0	140.0	140.0	0.0
craigslist.org	208.82.238.129	175.0	175.0	175.0	176.0	1.0
imgur.com	23.235.43.193	22.9	24.3	24.5	25.3	2.4
neobux.com	192.230.66.91	138.0	140.2	139.0	146.0	8.0
amazon.co.jp	54.240.250.0	-	-	-	-	-
t.co	199.16.156.75	116.0	116.9	117.0	119.0	3.0
reddit.com	95.100.96.11	23.1	30.9	27.9	52.1	29.0
xhamster.com	88.208.24.59	42.9	43.4	43.2	59.2	16.3
google.com.mx	81.175.29.152	0.8	0.9	0.9	1.2	0.4
stackoverflow.com	198.252.206.140	95.9	96.0	96.0	96.5	0.6
fc2.com	54.183.104.112	-	-	-	-	-
google.ca	81.175.29.177	0.8	0.9	0.9	1.4	0.6
bbc.co.uk	212.58.246.103	30.3	30.7	30.5	32.0	1.7
cnn.com	157.166.226.25	133.0	133.0	133.0	141.0	8.0
go.com	68.71.220.3	188.0	188.0	188.0	192.0	4.0
ifeng.com	210.51.19.61	250.0	250.1	250.0	256.0	6.0
aliexpress.com	205.204.96.1	185.0	185.4	185.0	186.0	1.0
xinhuanet.com	202.108.119.193	-	-	-	-	-
youku.com	121.9.204.234	256.0	256.0	256.0	259.0	3.0
vube.com	216.127.49.178	-	-	-	-	-
google.com.hk	173.194.65.94	30.6	30.7	30.7	30.8	0.2
blogger.com	173.194.65.191	29.8	29.9	29.9	30.1	0.3
alibaba.com	205.204.96.36	186.0	186.0	186.0	186.0	0.0
webcache.foreign.ccgslb.com	180.210.234.65	30.2	30.3	30.3	45.1	14.9
google.com.tr	81.175.29.163	0.8	0.9	0.9	2.5	1.6
odnoklassniki.ru	217.20.147.94	36.1	36.2	36.2	36.5	0.4
godaddy.com	97.74.104.201	175.0	176.2	176.0	201.0	26.0
huffingtonpost.com	205.188.101.58	-	-	-	-	-

Host	IP address	min	avg	median	max	max-min dist
kickass.to	62.210.141.210	40.7	41.3	41.3	41.6	0.9
pornhub.com	31.192.117.132	-	-	-	-	-
wordpress.org	66.155.40.249	182.0	183.0	183.0	183.0	1.0
thepiratebay.se	194.71.107.27	-	-	-	-	-
gmw.cn	124.207.134.2	-	-	-	-	-
google.com.au	81.175.29.177	0.8	0.9	0.9	2.5	1.7
amazon.de	178.236.6.250	-	-	-	-	-
adobe.com	192.150.16.117	147.0	147.0	147.0	150.0	3.0
ebay.de	66.135.215.61	-	-	-	-	-
google.pl	81.175.29.152	0.8	0.9	0.9	1.1	0.3
netflix.com	69.53.236.17	165.0	165.0	165.0	169.0	4.0
clkmon.com	108.168.157.82	140.0	140.0	140.0	150.0	10.0
dailymotion.com	195.8.215.138	36.1	37.2	37.3	39.5	3.4
chinadaily.com.cn	124.127.52.130	-	-	-	-	-
espn.gns.go.com	199.181.133.61	183.0	183.0	183.0	187.0	4.0
alipay.com	110.75.143.33	237.0	237.2	237.0	239.0	2.0
about.com	207.241.148.80	-	-	-	-	-
indiatimes.com	223.165.27.13	-	-	-	-	-
google.co.id	81.175.29.163	0.8	0.9	0.9	1.0	0.2
rakuten.co.jp	133.237.48.124	287.0	287.0	287.0	288.0	1.0
dailymail.co.uk	195.234.240.212	-	-	-	-	-
vimeo.com	107.162.132.45	111.0	111.0	111.0	115.0	4.0

Appendix D

Position paper

The following is a position paper that was published as part of the Internet Society Workshop on Reducing Internet Latency, September 2013. Note that we mixed up MFR and LT in the figure, and that it is *goodput* and not throughput that is plotted.

On the Treatment of Application-Limited Streams

Andreas Petlund[†], Anna Brunstrom[‡], Jonas Markussen[†], Markus Fuchs[‡*]
[†]Simula Research Laboratory, [‡]Karlstad University, [*]University of Kaiserslautern

Introduction

For streams that probe actively for bandwidth, a lot of work has been done to define how to behave fairly, mainly by dividing the resource of bottleneck throughput between the competing streams over time. Although this work targets greedy traffic it has a tendency to steer our thinking of what is fair for all types of traffic.

For streams that are application-limited, latency is more important than throughput. When reliable transport is required, the latency induced by the need to retransmit lost packets can cause problems. Redundancy and more aggressive retransmissions may improve latency for such streams. The use of aggression and redundancy has also been explored as a means of reducing retransmission latency [1, 2, 3]. However, to which degree such aggression should be allowed, and how to weigh the need for throughput against the need for latency, has only been superficially treated so far. Proposals that advocate more aggressive behaviours for application-limited streams are often met with scepticism and arguments that such behaviour will not be fair to competing traffic.

In this position paper, we present experimental results showing how application-limited streams lose against greedy streams in the "traditional", throughput-based fairness regime. Even when more aggressive retransmission mechanisms are applied, the application-limited streams still lose the battle against the greedy streams. Our results suggest that it is defendable to use aggressive retransmissions to reduce latency in many cases. We invite a discussion on how to define the level of aggression that can be applied without clogging the tubes and how to explore and formulate guidelines that help constantly application-limited streams recover in a timely fashion.

Sharing behaviour of application-limited streams

Figure 1 shows results from a set of experiments where we send an increasing number of streams over a 1Mbps bottleneck. The dotted line shows the expected bandwidth consumption for the thin streams if given their "fair share". We here define the fair share for the thin streams as the aggregate bandwidth needed for the thin streams as long as that number is less than half the bottleneck capacity. We also ran this experiment on thin streams using two mechanisms that increase aggressiveness for retransmissions: one where the sender performs a "fast retransmit" on the first dupACK it receives (mFR), thus reacting on the first indication that loss has happened; and one where six retransmissions using the base RTO are performed before the RTO is exponentially increased (LT), thus increasing the chance of recovering the segment without extreme delays. The two mechanisms were applied both individually and in combination. The goal of this experiment was to see if being slightly more aggressive will skew the throughput fairness in a severely congested scenario. The results show no significant difference between achieved throughput for competing greedy streams when aggressiveness for thin-stream retransmissions is raised. We can see in Figure 1 that, as competition gets tougher, the thin streams consistently lose the struggle for throughput-resources.

We focus in this position paper on the sharing characteristics of application limited streams, as arguments for why we believe more aggressive behaviours for such streams are well justified. The positive benefits on latency of the two more aggressive retransmission mechanisms used in the experiments were demonstrated in laboratory experiments in [2] and in a "live" game server evaluation in [4]

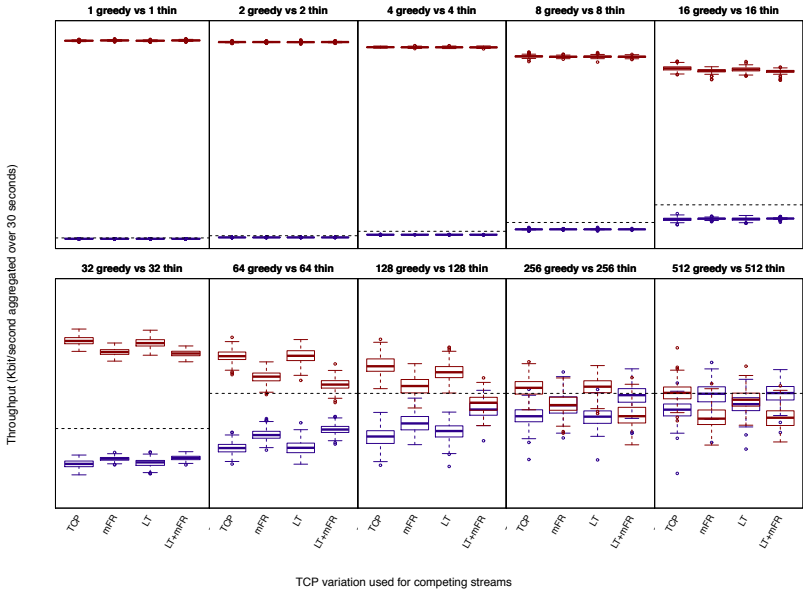


Figure 1: Aggregated throughput of thin streams competing with greedy TCP streams. The plot shows aggregate throughput of all greedy (red) streams and all thin streams (blue) normalised over 30 second intervals. Thin-stream properties: Packet-size: 160 B, Intertransmission-time: 150 ms. Link properties: Bottleneck bandwidth: 1Mbps, RTT: 100ms, queue size: 1 BDP (9 packets * 1500B)

Discussion

For congestion-controlled streams, fairness is deeply investigated, modelled and researched. Most commonly used when measuring fairness is Jain's fairness index [5], which is a general metric. Although it allows to assess fairness in different dimensions like latency or loss rate, throughput is still the predominant choice. For application-limited streams that are not aggressively probing for bandwidth, however, there is no clear consensus on how to limit aggressiveness. There has been a common practise of condemning spurious retransmissions in order to conserve bandwidth resources for goodput. Our view is that this argument should be reviewed when retransmission latency is important. Our experimental results show that application-limited streams are currently at a disadvantage when sharing resources with greedy streams, suggesting that more aggressive behaviour is appropriate.

Furthermore, we believe that improved mental models, as well as operational performance metrics, for how streams of different types should share network resources are required. It is unclear by which standards one should guide the evaluation of resource allocation in order to justly evaluate schemes based on redundancy and more aggressive retransmissions.

Acknowledgement

The authors are funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project(ICT-317700). The views expressed are solely those of the authors.

References

[1] M. Allman, K. Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)," RFC 5827 (Experimental), Internet Engineering Task Force, May 2010. [Online]. Available: http://www.ietf.org/rfc/rfc5827.txt

[2] A. Petlund, "Improving latency for interactive, thin-stream applications over reliable transport," Ph.D. dissertation, Simula Research Laboratory / University of Oslo, Unipub, Kristian Ottosens hus, Pb. 33 Blindern, 0313 Oslo, December 2009.

[3] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing Web Latency: the Virtue of Gentle Aggression," in *Proceedings of the ACM Conference of the Special Interest Group on Data Communication (SIGCOMM '13)*, 2013.

[4] A. Brunstrom, A. Petlund, and M. Rajiullah, "Reducing Internet Transport Latency for Thin Streams and Short Flows," in *Proceedings of Future Network and MobileSummit (poster paper)*, 2013.

[5] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, "A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems," DEC-TR-301, Digital Equipment Corporation, Tech. Rep., Sep. 1984. [Online]. Available: http://arxiv.org/abs/cs.NI/9809099